Advanced C++
Core Language and the STL

Jean-Paul RIGAULT

University of Nice - Sophia Antipolis
Engineering School — Computer Science Department
Sophia Antipolis, France

Email: jpr@polytech.unice.fr

Contents of the Course

Advanced C++
Core Language

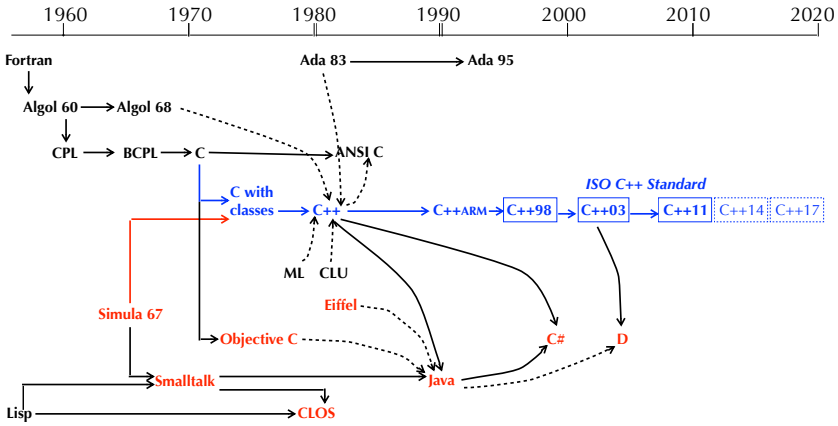
Part 1

Introduction

Characteristics of C++

- **Scandinavian school of object-oriented languages**
  - Hybrid language
    - Support procedural (imperative) style
    - Support object-oriented style
  - Compiled language
    - Maximal static analysis
    - Promote programming safety and run-time efficiency
- **Upward compatibility with ANSI C**

Characteristics of C++
Origin and influences



Characteristics of C++
Why is C++ difficult?

- **Language mechanisms are intrinsically complex**
  - Need to master the fundamental concepts of programming languages
  - An engineering trade-off, not an academic exercise
  - However, the language is consistent with its (clear) design philosophy
- **Object-orientation is a culture**
  - Stress on architecture
  - Need to revisit centralized and purely functional reflexes

Advanced C++
Core Language

Part 2

Basic C++ Concepts and Mechanisms

Basic C++ Concepts and Mechanisms
Summary

- **Objects and types**
- **The C++ class**
- **Exception basics**
- **Template basics**

Objects and Types
Types in C++

- **Types of variables and constants in C++**
  - Scalar types: Built-in types
    - Integral types: `char`, `int` and their variations, *pointers*
    - Real types: `float`, `double` and their variations…
  - Scalar types: Enumerations (user defined)
    - These are also *integral* types
  - Class types (user defined)
    - Structures and unions
    - Classes

Objects and Types
Instances, pointers, and references

- **Three ways of accessing an object**
  - Instance, pointer, and reference
  - All three available for all types (built-in or class)

Instance	Pointer	Reference

`int i = 1, j = 2;`
`string s1("hello");`
Copy duplicates the contents (the value)
`i = j;`
`string s2 = s1;`
	The value is an address
	`int *pi = &i;`
	Copy duplicates the contents (the value of the address)
	`int *pj = pi;`
	`pi = &j;`
	The *indirection operator* returns a *reference* to the pointed object
	`*pi = 3;`
		Another name for an object
		`int& ri = i;`
		No value semantics (*no copy of references*)
		`int& rj = j;`
		`ri = rj;`
		`// equivalent i = j`

Objects and Types
References (1)

- **References are mainly used for passing function parameters…**
  - to allow effective parameter modification within the function

```
void swap(int& a, int& b) {
  int tmp = b;
  b = a;
  a = tmp;
}

int x = 1, y = 2;
swap(x, y);
```

```
void swap(int *a, int* b) {
      int tmp = *b;
      *b = *a;
      *a = tmp;
}

int x = 1, y = 2;
swap(&x, &y);
```

  - or to avoid spurious copies

```
class Big_Stuff { … } bs; // expensive to copy
void print_Big_Stuff(const Big_Stuff&);
print_Big_Stuff(bs);
```


Objects and Types
References (2)

- **References can be used as function return type**
  - But beware of dangling references

```
string& f() {
  string st = "hello"; // local variable!
  return st;
}
```

  - Ensure we still reference something after the fonction returns

```
string& g(string& s) {
    if (...) return s;
  static string st = "hello"; // static local
  if (...) return st;
  else return *new string("bye");
 }
```

Objects and Types
Strong typing(1)

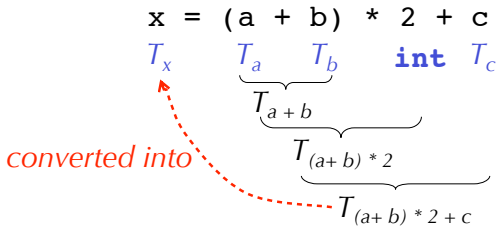
- **Every "object" has a unique type**
- **The type of an object must be known by the compiler when the object is used**
- **How is the type known?**
  - Either the object was declared in a type declaration

```
const int N = 100;          // constant
const string GREET = "hello";
string str = "hello";       // variable
int f(int, double);         // function prototype
int printf(const char *format, ...);
```

  - Or the type can be evaluated by the compiler
    - This is the case for *expressions* (next slide)

Objects and Types
Strong typing(2)

- **Type evaluation (for expressions)**
  - The type of an expression must be uniquely determined from the type of its components (operands)
    - The type evaluation is recursive w.r.t. sub-expressions
    - The context plays no role for evaluating an expression type



Objects and Types
Strong typing… but implicit conversions (1)

- **Implicit conversion**
  - A static (i.e., compile-time) conversion that the compiler can perform automatically when needed
- **Standard conversions**
  - Built-in conversions of C and Ansi C
    - Between various integer variations and enumerations
    - Between integers and real numbers
  - Built-in conversions of C++: inheritance conversion
    - From derived class to base class
    - For instances, pointers, and references

Objects and Types
Strong typing… but implicit conversions (2)

- **User defined (implicit) conversions**
  - From some type to a class
    - Defined by constructors with one parameter

```
string::string(const char *);
          // const char * → string
string s = "hello";
s = s + ", world!";// s = s + string(", world!")
```

  - From a class to some other type
    - Defined by the cast operator

```
string::operator const char *() const;
          // string → const char *
string form = "%d %d\n";
printf(form, i, j);
```

  - Beware of possible ambiguities!

Objects and Types
Strong typing… but implicit conversions (2)

- **User defined conversions**
  - From some type to a class
    - Defined by constructors with one parameter

```
string::string(const char *);
          // const char * → string
string s = "hello";
s = s + ", world!";// s = s + string(", world!")
```

  - From a class to some other type
    - Defined by the cast operator

```
string::operator const char *() const;
          // string → const char *
string form = "%d %d\n";
printf(form, i, j);     printf(form.c_str(), i, j);
```

  - Beware of possible ambiguities!

Objects and Types
Overloading

- **Overloading**
  - Same name for different functions

```
double cos(double);
complex cos(complex);
```

  - The functions differ by their parameters (number and/or types)
  - The return value type plays no role
    - Distinguishing two functions only from their return value type would break the rule that says that the type of an expression is uniquely determined from the type of its operands

```
int f();
double f();
int x = f(); // which type for expression f()?
```

- **Overloading resolution:** *see Part 3 (Templates)*

The C++ Class
What's in a class?

- **C programmer**
  - A "natural" extension of C structures (struct)
- **OO programmer/modeler**
  - A model for creating/deleting instances
  - The set of its instances (the class extension)
  - An object
- **Programming language specialist**
  - An (abstract) data type
- **Software engineer**
  - A name space
  - An encapsulation unit
  - A reuse unit

The C++ Class
What's in a class? C++ view

- **C programmer**
  - A "natural" extension of C structures (`struct`)
- **OO programmer/modeler**
  - A model for creating/deleting instances — *memory allocation initialization/finalization destructor/constructor*
  - The set of its instances (the class extension)
  - An object (despite RTTI… and templates)
- **Programming language specialist**
  - An (abstract) data type — *class utilization copy operations member-functions operator overloading*
- **Software engineer**
  - A name space — *see **namespace** further*
  - An encapsulation unit — *remind beginner's C++*
  - A reuse unit — *see **inheritance** further*

C++ Class: Extension of C Structure
Underlying C structure

- **A C++ class adds many features to a C structure**
  - Member-functions
  - Access control
  - Initialization/finalization operations
  - Inheritance
- **However, any C++ class has an underlying C structure**
  - Composed of the class instance data members
  - Representing the memory layout of the class instances
  - The size of which is the size of the class (`sizeof`)
  - Of course, the C++ programmer cannot manipulate this structure directly…

C++ Class: Model for Creating instances
Creating/deleting objects in C++

- **Object creation**
  1. Allocating memory
     - 4 different ways
  2. Initializing the object
     - The role of a class constructor
- **Object deletion**
  1. Finalizing the object
     - The role of the class destructor
  2. Returning memory
- **The two steps cannot be dissociated for C++ class instances**

C++ Class: Model for Creating instances
Memory allocation, lifetime, scope (1)

	Where?	Lifetime	Scope
Automatic objects	Stack	Block automatic deletion	Block
Statically allocated objects	Data segment	Whole program execution automatic deletion	Block (`static`) Class (`static`) File (`static`) Whole program (`extern`)
Dynamically allocated objects	Heap	From `new` to `delete` no automatic deletion	*Scope of the pointers that give access to them*
Exception objects	Heap?	From `throw` to last `catch` automatic deletion	Catch clauses

C++ Class: Model for Creating instances
Memory allocation, lifetime, scope (2)

```
// File f1.cpp
extern int glob;
static double PI = 3.14;

string *f() {
  double x;
  static int st = 10;
  if (...) throw Exc();
  return new string();
}
```

automatically allocated
statically allocated
dynamically allocated
thrown (exception)

```
// File f2.cpp
extern string *f();
int glob = 12;

void g() {
  try {
    string *ps = f();
    // ...
    delete ps;
  }
  catch (Exc& e) {
    cout << e.what() << endl;
  }
}
```

C++ Class: Model for Creating instances
Constructors: guarantee of initialization (1)

- **If a class has constructors, one of them will be invoked each time an instance is created**
  - Just after memory allocation
- **If the constructor requires parameters they must be provided when creating objects**
- **If an instance is constructed without any constructor parameters, its class must be constructible by default**
  - Either it has a default constructor (a constructor without any parameters)
  - Or it has no constructors at all and all its base classes and all its data members must be constructible by default (see next slide)

C++ Class: Model for Creating instances
Constructors: guarantee of initialization (2)

- **Default default constructor**
  - When a class has no constructors, the C++ compiler tries to synthesize a default constructor, by default
  - The process may fail… leading to a compile-time error

```
class A {
private:
  A() {}
};

class B {
public:
  B(int);
};
```

```
class C : public A {
private:
  B _b;
};

C::C() : A(), _b()
      // impossible!
{}
```


C++ Class: Model for Creating instances
Constructors: guarantee of initialization (3)

- **All created objects have the guarantee of initialization (call to one constructor)**
  - Static objects are created at the start of program execution
    - C++ does not define any order for static initialization across several compilation units
  - Automatic objects are created at the point where they are defined in their block
  - Dynamic objects are created by a call to `new`
  - Exception objects are created by a call to `throw`

C++ Class: Model for Creating instances
Destructor: finalizing objects(1)

- **All objects have the guarantee of finalization (call to the destructor) except dynamically allocated ones**
  - Static objects are destroyed at the end of the program execution (exit())
    - C++ does not define any order for static destruction across several compilation units
  - Automatic objects are destroyed at the end of their block of definition
  - Exception objects are destroyed when they are no longer needed (end of the last catch clause)
  - Dynamic objects must be deleted by explicitly calling `delete`
- **Destruction occurs in reverse order of creation**

C++ Class: Model for Creating instances
Destructor: finalizing objects(2)

- **Default destructor**
  - If a class has no destructor, the C++ compiler will try to synthesize a default destructor
    - Call the destructors of own data members
    - Call the destructors of all base classes

C++ Class: Model for Creating instances
Construction/destruction order (1)

- **Construction order (for complex object)**
  1. Construction (initialization) of base classe(s)
  2. Construction (initialization) of class members
     - The data members that the class adds (to the ones that it inherits)
  3. Constructor body
- **Destruction order (inverse order)**
  1. Destructor body
  2. Destruction (finalization) of class own members
  3. Destruction (finalization) of base classe(s)

C++ Class: Model for Creating instances
Construction/destruction order (2)

- **Base classes are constructed in the order of the derivation list**
- **Data members are constructed in the order of their declaration in the class definition**
  - Attention! DANGER !

```
class B : public A1, public A2 {
private:
  int *pt;
  int dim;
public:
  B(int n) : dim(n), A1(n), pt(new int[dim]) {...}
      // pt(new int[n]) would be correct!
};
```


C++ Class: Model for Creating instances
Construction/destruction order (3)

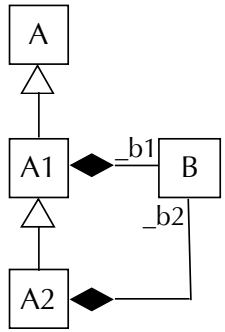
```
class B {public: B(int = 0);};

class A {private: int _p;
public:
  A(int = 0);
  // ...
};

class A1 : public A {
protected: B _b1;
public:
  A1(int i = 0, int j = 1)
    : _b1(i), A(j) {...④}
  // ...
};

class A2 : public A1 {private: _b2;};

A1 a1(2, 3);
```



C++ Class: Model for Creating instances Construction/destruction order (4)

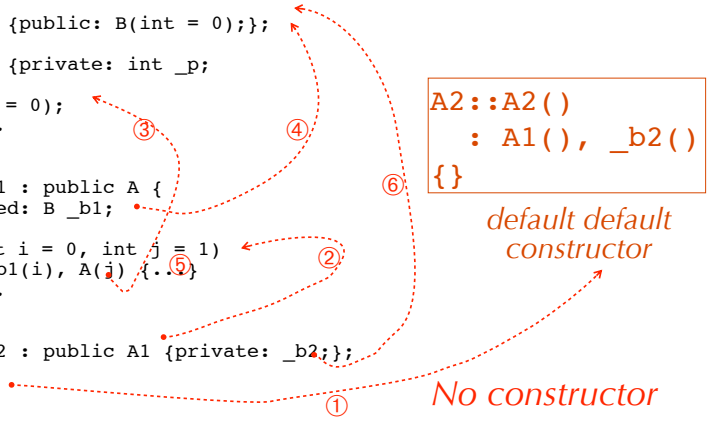
```
class B {public: B(int = 0);};

class A {private: int _p;
public:
  A(int = 0);
  // ...
};

class A1 : public A {
protected: B _b1;
public:
  A1(int i = 0, int j = 1)
    : _b1(i), A(j) {...}
  // ...
};

class A2 : public A1 {private: _b2;};

A2 a2;
```




```
A2::A2()
  : A1(), _b2()
{}
```

default default constructor

No constructor

C++ Class: Abstract Data Type

- **Set of values and operations representing an abstraction of an application domain entity**
  - Underlying C structure: the values
  - Member and friend functions: the operations
- **Features common to all instances: static members**
- **Regard instances as first class citizens**
  - Instances, with value semantics if needed
    - Copy operations can be user defined
  - All type constructions: constant instances, pointers and references to instances, arrays of instances…
  - Possibility of defining operators (expression syntax)

C++ Class: Abstract Data Type Defining operators for a class (1)

- **Operator overloading does not modify expression parsing**
  - No new notation for operators
  - No modification of precedence or associativity
  - No modification of the number of operands
- **No redefinition for built-in types**
  - At least one parameter must have class type
- **All operators are redefinable, except**
  - Preprocessor operators: `#` `##`
  - "Compile-time" operators: `typeid` `sizeof` `::`
  - The three following operators: `.` `.*` `?:`
    - However, note that `->` and `->*` can be redefined

C++ Class: Abstract Data Type Defining operators for a class (2)

- **To overload an operator**
  - Free function operator

```
string operator+(const string& s1, const string& s2);
string str = "hello";
string str1 = str + ", world"; // implicit conversion
                               // operator+(str, ", world")
```

  - Member-function operator

```
class A {
public:
   A(const string& s);
   A operator+(const A& a); // A::operator+
   // ...
};
A a("hello");
A a1 = a + ", world"; // a.operator+(A(", world"));
A a1 = ", world" + a; // no conversion on first param
```

C++ Class: Abstract Data Type
Copy operations (1)

- **Copy during initalization**

```
A a1(…);
A a2 = a1;  // this is not an assignment
```

- **Copy constructor**

```
A::A(const A&);
```

  - Invoked by default each time an instance of `A` is initialized from another instance of `A`
    - Simple initialization
    - Passing parameters and returning from functions by value
    - Default copy of classes with no copy constructor (see later)

C++ Class: Abstract Data Type
Copy operations (2)

- **Copy during assignment**

```
A a1(…), a2(…);
a2 = a1;
```

- **Copy assignment operator**

```
A& A::operator=(const A&);
```

  - Return type may vary (`void`, `A&`)
  - Invoked by default each time an instance of `A` is assigned to another instance of `A`
    - Simple assignment
    - Default copy of classes with no copy assignment operator (see later)

C++ Class: Abstract Data Type
Copy operations (3)

- **An assignment is not an initialization**
  - There is exactly one initialization throughout the entire life of an object
  - Initialization is always possible (once) whereas assignment may be forbidden (const objects)
  - Assignment must take care of the previous state of the object whereas, for initialization, there is no such thing as a previous state
- **An assignment trap:**
  - Beware when writing the assignment operator: `x = x;`
- **Copy assignment may often be defined in term of copy construction**

C++ Class: Abstract Data Type
Copy operations: default copy (1)

- **If a class has copy constructor(s) and/or assignment operator, these functions are entirely responsible for copying objects**
- **Otherwise, *default copy* is performed memberwise**
  - The C++ compiler synthesizes the needed copy operations (default copy constructor, default copy assignment operator)
    - Each member is copied according to its own copy semantics
    - Base class(es) are considered as members during the copy operation
    - The memberwise procedure is applied recursively
    - Built-in types are copied bitwise
  - The synthesis process may fail…

C++ Class: Abstract Data Type
Copy operations: default copy (2)

```
class B : public A {
  int i;
  char *pc;
  string s;
  // no copy operations
};

B b1(…);
B b2 = b1;
b1 = b2;
```

```
B::B(const B& b)
  : A((const A&)b),
    i(b.i), pc(b.pc), s(b.s)
{}
```

```
B& B::operator=(const B& b) {
  *(A*)this = (const A&)b;//!!
  i = b.i;
  pc = b.pc;
  s = b.s;
  return *this;
}
```

- Note that i *and* pc are bitwise copied

C++ Class: Abstract Data Type
Copy operations: default copy (3)

- **Synthesis failure of default copy operations**

```
class A {
private:
  const string s; // const member
  B& rb;          // reference data member
  A(const A&);    // private copy constructor
};
```

  - The `const` member or the reference prevent the synthesis of the default copy assignment (but *not* of the default copy constructor)
  - The private copy constructor prevents the synthesis of the copy constructor for any class that contains an `A` by value or that derives from `A`

C++ Class: Abstract Data Type
Copy operations: non copiable objects

- **To forbid copying objects for a class, make copy operations (copy constructor and copy assignment operator) private**
  - Of course, this forbids copy only outside the class
  - A bullet-proof solution is to declare the copy operations but not to implement them: any attempt to copy will lead to a link edition error…

```
class Non_Copiable {
private:
  Non_Copiable(const Non_Copiable&);
  Non_Copiable& operator=(const Non_Copiable&);
     // No definition for these operations
public:
  Non_Copiable() {} // needed here!
};
```

  - Obsolete in C++ 2011!

C++ Class: Name Space

- See **PART 3 : Namespaces**

C++ Class: Encapsulation Unit
Access control

- **Motivations for access control**
  - Security: not in C++?
  - Reducing dependencies
  - Preserving class internal consistency
  - *"C++ access control does not prevent from cheating"* (B. Stroustrup)
- **Access control is on the basis of the class**
  - A class instance has access to the private members of all instances of the same class
  - There is a special case for protected members (see later)

C++ Class: Encapsulation Unit
Access control levels

- **Three levels of access control for members**
  - **`private`**: members exclusively private to the class
  - **`protected`**: member private to the class and its derivatives
  - **`public`**: members visible and manipulable within any context
- **Three levels of derivation: inherited member level in the derived class**
  - **`private`**: `protected` and `public` members → `private`
  - **`protected`**: `public` members → `protected`
  - **`public`**: `protected` and `public` members remain so
- **Default is `private` for `class`, public for `struct`**

C++ Class: Encapsulation Unit
Friendship (1)

- **Friend function**
  - A function that is not a member of the class
  - But that has access to the class private and protected members
- **Friend class**
  - All the friend class member functions are friend of the first class
- **Restriction on friendship**
  - Not symmetric
    - A friend of `B` does not imply `B` friend of `A`
  - Not transitive
    - Friends of my friend are not my friends
  - Not inherited
    - Parents' friends are not children's friends
    - Children's friends are not parents' friends
  - It is impossible to impose a friend to a class

C++ Class: Encapsulation Unit
Friendship (2)

```
class A {
  friend void f(A);
  friend class B;
private:
  int _priv;
protected:
  int _prot;
};

void f(A a) {
  A a1;
  a1._priv = a._priv;
}
```

```
class B {
  friend void g();
private:
  int _i;
public:
  void m(A a) {
    A a1;
    a1._priv = a._priv;
  }
};

void g() {
  A a;
  _i = a._priv;    // NO
}
```

Exception Basics
Motivation

- **Exceptions are mandatory because of encapsulation**
- **What can be done when an abnormal condition is detected inside a class?**
  - Notify the user
  - Allow her to recover and resume the program at a suitable point
  - Separate normal processing from exceptional one
  - Respect the C++ creation/deletion semantics
    - C functions `setjmp()`/`longjmp()` are not appropriate because local objects with destructor must be destroyed properly

Exception Basics
Basic mechanism

```
main() {f();}

void f() {
  try {
      g();
  }
  catch (E1) {...};
  catch (E2) {...}
}

void g() {
  try {
      h();
  }
  catch (E1) {...}
}

void h() {
  if (...) throw E1();
  else throw E2();
}
```

- **An exception is a temporary object (of any copiable type)**
  - it is distinguished only by the way it is created (`throw`)
- **When an exception is raised (thrown) the current function is exited, and the exception is propagated upward, traversing (and exiting) functions in the calling chain**
- **The propagation is stopped by the first try block with a catch for the (type of) the exception**
- **The catch clause (handler) is executed;**
- **If the handler does nothing special, control is then transferred after the try block (and all its catch clauses)**

Exception Basics
Basic mechanism: destruction of locals

```
main() {f();}

void f() {
  try {
      string s1;
      g();
  }
  catch (E1) {...};
  catch (E2) {...}
}

void g() {
  try {
      string s2;
      h();
  }
  catch (E1) {...}
}

void h() {
  string s3;
  if (...) throw E1();
  else throw E2();
}
```

destruction of s1
destruction of s2
destruction of s2
destruction of s3
destruction of s3

- **If a function is abandonned or traversed by an exception, all automatic (local) objects are properly destroyed (i.e., their C++ destructor is called)**
- **Nothing similar occurs for dynamically allocated objects**
- **Thus it is the programmer's responsability to provide finalization for dynamically allocated objects**
  - See RAII later

Exception Basics
Basic mechanism: rethrowing an exception

```
main() {f();}

void f() {
  try {
      ...
      g();
  }
  catch (E1) {...};
}

void g() {
  try {
      ...
      h();
  }
  catch (E1) {...; throw;}
}

void h() {
  if (...) throw E1();
}
```

rethrow

- **An exception can be caught, then rethrown to continue its propagation upwards (the call chain)**

Exception Basics
Basic mechanism: uncaught exception

- **If no try block in the calling chain has a catch clause for the current exception, the standard function `terminate()` is called**
- **By default, `terminate()` should simply call `abort()`**
  - In recent compilers, `terminate()` also prints out the name of the exception that was received
- **The user may provide her own version of `terminate()`**

Template Basics
General picture

- **Possibility to create functions and classes templates parameterized (mainly) with types**
- **Two different mechanisms for functions and classes**
  - Although somewhat homogenized by ANSI C++
- **Static resolution (compile and link time)**
  - Thus the template parameters must be static entities
  - A type driven macro-processing facility

Template Basics
Function template: definition

- **Only one representation of identical operations, but on different types**

```
template <typename T>
inline T Min(T a, T b) {
  return a < b ? a : b;
}
```

  - Constraints on template parameters are implicit ("duck typing")
    - `T` denotes here any C++ type (class or built-in)…
    - … as long as the type has `operator<` and a copy constructor (possibly the default one)

Template Basics
Function template: usage

- **Instantiation of function templates is generally implicit**

```
int i, j;
double x, y;
j = Min(i, 3);  // int Min(int, int)
y = Min(x, y);  // double Min(double, double)
```

  - Implicit instantiation (function code generation) is within the responsibility of overloading resolution
    - a parameterized overloaded function

Template Basics
Function template: overloading with regular function

- **The previous form of `Min` is not suitable for C nul-terminated character strings (`char *`)**
- **Of course, the programmer may provide a regular function that will supersede template instantiation**
  - because "most exact" match precedes template instantiation during overloading resolution

```
const char *Min(const char *s1, const char *s2) {
  return strcmp(s1, s2) < 0 ? s1 : s2;
}
```

  - Note that, technically, this is different from *template specialization*, it is just overloading

Template Basics
Class template: definition (1)

- **One unique representation of a set of classes that differ only by the types of objects they manipulate or possibly by some constants**

```
template <typename Elem> class List {...};

template <typename Elem, int N>
class Fixed_Array
{
  Elem tab[N]; // N must be a compile-time constant
  // ...
};
```


Template Basics
Class template: definition (2)

```
template <typename Elem>
class List {
public:
  List();
  void append(const Elem& e);
  void prepend(const Elem& e);
  // ...
};
```

```
template <typename Elem>
void List<Elem>::List() {
  // ...
}

template <typename Elem>
void List<Elem>::append
          (const Elem& e) {
  // ...
}

template <typename Elem>
void List<Elem>::prepend
          (const Elem& e) {
  // ...
}
```


Template Basics
Class template: instantiation

- **Instantiation of class template must be explicit**

```
List<int> l1;
List<List<int> > l2; // attention: > >

Fixed_Array<double, 1000> fa1;
Fixed_Array<List<int>, 100> fa2;

List<Fixed_Array<List<char>, 10> > l3;

typedef List<int> List_int;
List_int l4;
```

Advanced C++
Core Language

Part 3

Advanced C++ Mechanisms

Advanced C++ Mechanisms
Summary

- **Advanced exceptions**
- **Overloading resolution**
- **Complements on function and class templates**
- **Run Time Type Information (RTTI) and ISO C++ casts**
- **Multiple inheritance**
- **Name packaging (namespace)**

Advanced Exceptions
Organizing exceptions: catch resolution(1)

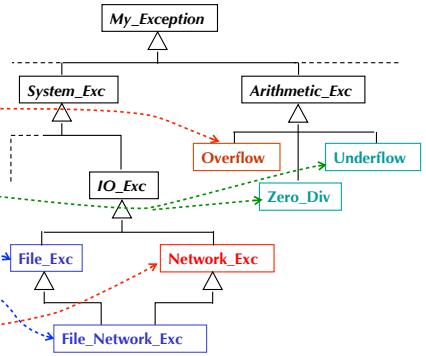
- **For a same try block, catch clauses are scanned from top to bottom**
- **Type conformance must be exact, except that standard conversions related to inheritance (only) are authorized**
- **The temporary object created by throw may be retrieved in catch, by value but also by reference**

```
catch (Stack::exception& e) {cout << e.what();}
```

  - Above, polymorphism may be honored for `e`
  - It is a good idea to always *catch by reference*
- **The form `catch (...)` captures all exceptions**
  - When needed, it should be the last catch clause

Advanced Exceptions
Organizing exceptions: catch resolution(2)

```
try {
  // ...
}
catch (Overflow& ov) { }
catch (Arithmetic_Exc&
e) {
  cout << e.what();
}
catch (File_Exc) { }
catch (IO_Exc) { }
catch (...) {
  cout << "unexpected";
}
```



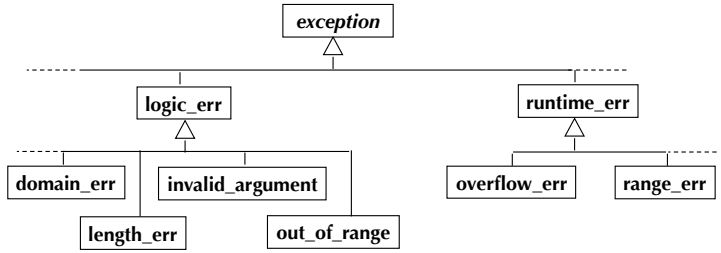
Advanced Exceptions
Organizing exceptions: `exception` (1)

- **Defined in the ISO standard**

```
class exception {
public:
 exception() throw ();
 exception(const exception&) throw ();
 exception& operator=(const exception&) throw ();
 virtual ~exception() throw ();
 virtual const char *what() const throw ();
};
```

  - `throw()` in C++03 (`noexcept` in C++11) means that the function should not throw exceptions

Advanced Exceptions
Organizing exceptions: `exception` (2)



- **The most important for an exception is a *significant name***

```
class Null_Pointer_Exc {};
```

- **It is not mandatory to use the standard exception hierarchy**
- **One is free to define a specific exception inheritance tree with dedicated mechanisms**

Advanced Exceptions
Resource Acquisition is Initialization (1)

`void f1()`
`{`
`  FILE *fp = fopen("toto", "w");`
`  // work with fp`
`  fclose(fp);`
`}`
	Who is going to call `fclose` if an exception traverses this function?

`void f2()`
`{`
`  A *p = new A();`
`  // work with p`
`  delete p;`
`}`
	Who is going to `delete p` if an exception traverses this function?

`void f3()`
`{`
`  lock(&mutex);`
`  // work in critical section`
`  unlock(&mutex);`
`}`
	Who is going to release the mutual exclusion lock (`unlock(&mutex)`) if an exception traverses this function?

Advanced Exceptions
Resource Acquisition is Initialization (2)

`class My_File {`
`  FILE *_fp;`
`public:`
`  My_File(...) {_fp = fopen(...);}`
`  ~My_File() {fclose(_fp);}`
`  // ... accessors to stdio functions`
`};`
	`void f1()`
	`{`
	`  My_File mf("toto", "r");`
	`  // work with mf as if a FILE*`
	`  // No need to close`
	`}`

`class My_Handle {`
`  A *_p;`
`public:`
`  My_Handle(A *p) _p(p) {}`
`  ~My_Handle() {delete _p;}`
`  // ... accessors to *_p`
`};`
	`void f2()`
	`{`
	`  My_Handle h(...);`
	`  // work with h as if pointer`
	`  // No need to delete`
	`}`

`class My_Lock {`
`  Mutex *_m;`
`public:`
`  My_Lock(Mutex *m) :_m(m) {lock(_m);}`
`  ~My_Lock() {unlock(_m);}`
`};`
	`void f3()`
	`{`
	`  My_Lock lock(&mutex);`
	`  // work in critical section`
	`  // No need to unlock`
	`}`

Advanced Exceptions
Resource Acquisition is Initialization (3)

- **Library classes to implement RAII**
  - `My_File`? Use `iostreams`
  - `My_Lock`? All thread libraries have a standard lock class
  - `My_Handle`? Use RAII standard smart pointers
- **RAII smart pointers**
  - They are substitutable for ordinary pointers
  - Their destructor destroys the object pointed to when needed
    - `std::auto_ptr`: for C++98 and C++03, bizarre copy, obsolete
    - `std::unique_ptr`: for TR1 and C++11, not copiable
    - `std::shared_ptr`: for TR1 and C++11, copiable, safe sharing
    - `std::weak_ptr` to help `std::shared_ptr` cope with circularity

Advanced Exceptions
Resource Acquisition is Initialization (4)

- **Class `auto_ptr<T>`**
  - Used to implement *Resource Acquisition is Initialization* (RAII)

```
#include <memory>
void f() {
  auto_ptr<A> ap(new A(...));
  // use ap as a pointer
  // object pointed to automatically deleted
}
```

  - Constructor is explicit

```
auto_ptr<A> ap = new A(...); // Forbidden
```

  - It is forbidden to assign a regular pointer to an `auto_ptr`
  - "Special" copy operations…

Advanced Exceptions
Resource Acquisition is Initialization (5)

- **Copy of `auto_ptr`: ownership transfer**
  - When copying auto pointers, ownership of the object pointed to is transferred to the target
  - The object previously owned by the target, if any, is deleted
  - The previous owner (source of the copy) looses ownership and its internal pointer is set to null
  - Thus copying or passing by value modifies the source of the copy
    - This is a violation of usual rules
    - Now `auto_ptr` are obsolete

```
auto_ptr<A> ap1(new A());
void f(auto_ptr<A> ap) {
  // ...
  // object owned by ap deleted
}

ap1 = auto_ptr<A>(new A());
  // Object previously owned by
  // ap1 is deleted and ap1
  // becomes owner of new one
f(ap1);
  // ap1 has lost ownership and
  // is now null!!
```


Advanced Exceptions
Throwing exception in a constructor (1)

- **All members fully constructed before the exception is thrown are deleted**
  - The destructors of fully constructed components are correctly called
  - Partially constructed sub-objects are not destructed
- **The object itself has not been constructed, thus its destructor won't get called**
- **It is indispensable to understand clearly in which order constructions/deletions are performed**
- **It is not a good idea to throw an exception from within a destructor!**

Advanced Exceptions
Throwing exception in a constructor (2)

```
class A1 {
  string id;
public:
  A1(const string& s) : id(s) {}
};

class A2 {
  int val;
public:
  A2(int v) {
      if (v == 0) throw Exc_A2();
  }
};

class B : public A1, public A2 {
  string name;
public:
  B(const string& s, int v)
      : A1(s), A2(v) {
      name = s; throw Exc_B();
  }
};
```

Fully constructed parts

	v = 0	v ≠ 0
~A1()	✔	✔
~A2()	✗	✔
name (~string())	✗	✔
~B()	✗	✗

Advanced Exceptions
Exception-safe code(1)

- **Basic guarantee**
  - Don't leak resources in case of exceptions
  - The program should be left in consistent (but not necessarily predictable) state
- **Strong guarantee**
  - If an operation terminates because of an exception, program state is exactly the same as before the operation started
  - Sort of commit and roll-back strategy
- **Nothrow guarantee**
  - Some operation should guarantee to throw no exception
    - e.g., destructors, `operator delete`...

Advanced Exceptions
Exception-safe code(2)

- **Some tips**
  - Always use RAII when applicable, to avoid leaking resources
  - Do not return a value from a mutator member-function
  - Try to define related operators in an homogeneous way
    - Implement copy assignment from copy constructor
    - Operator += from + and =
    - Relational operators from operator < and ==
    - etc.
  - In a function, isolate the code that may throw an exception and do the work off to the side, before committing

Overloading Resolution
General principle (1)

- **Purpose of overloading resolution**
  - Given a function call

$$\ldots\ \texttt{f}(a_1, a_2, \ldots, a_n)\ \ldots$$

  - find a function with the same name matching the call
    - `n` is known, as well as the argument types T_1, T_2, ..., T_n
    - Context and return type play no role
  - The compilers can determine
    - the number and types of the effective parameters
    - the prototypes of all functions with the same name `f` visible at the point of call

Overloading Resolution
General principle (2)

1. **Identify candidate functions**
   - All functions visible at the point of call and bearing the same name as the one that is called
   - Relies on name lookup
   - Special processing for function templates (see later)
2. **Among candidate functions, select the viable ones**
   - All candidate functions that can be considered for the call
   - Functions should have adequate number of arguments
   - There should exist conversions from each effective parameter to the corresponding formal parameter of a viable function
3. **Among viable functions, select the unique best match if it exists**
   - Relies on ranking possible implicit argument conversions

Overloading Resolution
General principle (3)

- **Ranking implicit conversion for each parameter**
  - Exact match:
    - strict exact match (identity)
    - *lvalue* transformations
  - Qualification adjustment: adding const or volatile
  - Promotions: integral and floating point
  - Other standard conversions:
    - integral and floating point conversions
    - pointer conversions
    - inheritance conversions
  - User defined conversions: constructor, cast operator
    - at most one for each argument

Overloading Resolution
General principle (7)

- **The best match viable function**
  - The conversions applied to its arguments are no worse than the conversions necessary for any other viable function
  - There is at least one argument for which the conversion is better than the conversion necessary for the same argument when calling the other viable function

Overloading Resolution
Function template: argument deduction (1)

- **Overloading resolution with function templates as candidates**
  - Which function template instances to chose?
- **Argument deduction**
  - The process of matching the call arguments with the function template parameter types
  - Determine with function to instantiate
    - Part of overloading resolution
- **When the type of a parameter is a template parameter, only the following conversions are accepted**
  - Exact match
  - Qualification adjustment
  - Derived to base (inheritance) conversions
- **It is not an error for argument deduction to fail (SFINAE)**

Overloading Resolution
Function template: argument deduction (2)

```
template <typename T>
const T& Another_Min(const T&, const T&);

const string sc1, sc2;
string s = Another_Min(sc1, sc2);  // identity match

string s1, s2;
s = Another_Min(s1, s2);    // qualification
adjustment

double x = Another_Min(3, 7.5);
    // no acceptable conversion for argument
deduction
```


Overloading Resolution
Function template: argument deduction (3)

- **Modification of overloading resolution**
  - Candidate functions
    - Include possible instances of function template for which argument deduction succeeds
    - If there is a template specialization for the corresponding argument deduction, include it instead of the primary template instance
  - Viable functions
    - Unchanged…
    - … but candidates issued from template instantiation are automatically viable
  - Best match
    - Prefer non-template over template instance in case of equal ranking

Complements on Templates
Explicit instantiation of function templates

- **Needed when the compiler cannot guess how to instantiate the template**

```
template <typename T, int N> T f();
f();   // which value for N? what is T?
f<int, 10>();   // OK
```

  - Once instantiated, we have a regular function with all usual conversions

```
double x = Min(5, 7.2); // no match
double x = Min<double>(5, 7.2); // OK
```

- **Beware: possible syntactic ambiguities**

```
template <int N> void f(int);
i = f<3>(j); // it can read as: (f<3) > (j)
i = template f<3>(j); // OK, but nasty!
```


Complements on Templates
Class template and class derivation

- **Both mechanisms are totally compatible**

```
template <typename T>
class A {...};

template <typename T>
class B : public A<T> {...};

template <typename T>
class Shared : public T {int refcnt; ...};

class C : public A<int> {...};
template <typename T>
class D : public C {...};

template <typename T>
class MyClass : Serializable<MyClass> {...}; // CRTP
```

Complements on Templates
Class template and implicit conversions (1)

- **Two instances of the same class template define different types as soon as their effective template parameters differ**
- **Template instances and implicit conversion**
  - Since there exists an implicit conversion from `int` to `double`, shouldn't a `List<int>` be implicitly convertible into a `List<double>`?
  - Since a French is a Human (suppose), shouldn't a set of French be implicitly convertible into a set of Human?
  - The answer is NO in both cases
  - Should the answer be yes, substitutability principle, contravariance rule, and static type checking would be broken...

Complements on Templates
Class template and implicit conversions (2)

- **Breaking static type checking**
  - Assume French (and English) derives from Human
    - Hence, `French *` is implicitly convertible into `Human *`
  - Suppose we allow ***implicit conversion*** from `set<French *>` to `set<Human *>`
  - Static typing is broken!

```
void f(set<Human *>& sh) {
  sh.insert(new English());
    // OK since English are
    // Human, after all
}

set<French *> sf;
// ...
f(sf);// implicit conversion
  // There is now an English
  // among the French!
```


Complements on Templates
Class template and static data members

- **Initialisation of static members of class template**

```
template<typename T> class A {
 static int _i;
 static T _s;
 // ...
};
template <typename T> int A<T>::_i = 0;
template <typename T> T A<T>::_s = T();
```

  - The initializations must be in the same scope as the definition of the class
  - Omitting initialization causes a link-time error
    - but only if the static members are used!

Complements on Templates
Class template: member template(1)

- **Member template of a regular class**

```
class A {
public:
  template <typename U> void mt(U u);
};
template <typename U> void A::mt(U u) { ... }
```

  - Just a parametrized overloaded member function

```
A a;
a.mt(3);        // A::mt(int)
a.mt("hello"); // A::mt(char *)
a.mt(3.5);      // A::mt(double)
```

Complements on Templates
Class template: member template(2)

- **Member template of a class template**

```
template <typename T>
class A {
public:
 template <typename U> void mt(U u);
};

template <typename T>
template <typename U>
void A<T>::mt(U u) { ... }

A<int> a;
a.mt("hello"); // A<int>::mt(char *)
```

- **Member templates cannot be virtual**

Complements on Templates
Class template: friend function template

- **Friend function template of a regular class**

```
class A {
  template <typename U> friend void tf(U u);
  // ...
};
template <typename U> void tf(U u, ...) { ... }
```

- **Friend function template of a class template**

```
template <typename T>
class A {
  template <typename U> friend void tf(U u) {
    ...
  } // must be defined inline!
  // ...
};
```


Complements on Templates
Class template: friend class template

- **A class template can be friend of a regular class as well as of a class template**

```
class A {
  template <typename>
  friend class F;
private:
  int _i;
};

template <typename T>
class B {
  template <typename>
  friend class F;
private:
  T _t;
};
```

```
template <typename U> class F {
public:
  void m() {
    A a;
    a._i = 12;  // OK
    B<int> b1;
    b1._t = 6;  // OK
    B<U> b2;
    b2._t = U(); // OK
  }
};

int main() {
  F<double> fr;
  fr.m(); // OK
}
```


Complements on Templates
Template parameters of templates (1)

- **ISO C++ unifies the template parameter possibilities for template functions and classes**
- **A generic parameter may be**
  - A type (built-in or class)
  - A static constant of any integral type
    (this includes enumerations and also pointers to members and pointers to **extern** variables and functions
  - Another class template
- **Default values are possible for template parameters**
  - But for class templates only, not for function templates (in C++03)

Complements on Templates
Template parameters of templates (2)

- **Pointers as template parameters**

```
template <typename T, void (*SORT)(int, T[])>
class Sortable {...};

extern void quicksort(int, double[]);
Sortable<double, quicksort> s(...);
```

  - Allow to parameterize internal implementation (algorithms)
  - The pointed object must be `extern`

Complements on Templates
Template parameters of templates (3)

- **Default values for template parameters**

```
template <typename T = int, int N = 10>
class Fixed_Array {...};

Fixed_Array<double, 100> fa1;
Fixed_Array<double> fa2; // Fixed_Array<double,
10>
Fixed_Array<> fa3;       // Fixed_Array<int, 10>
```

  - Rules are similar to default value for function parameters

Complements on Templates
Template parameters of templates (4)

- **Class templates as template parameters**

```
template < typename U,
           template <typename, typename>
           class Container = std::vector >
class Something {
  Container<string, std::allocator<string> > _cs;
  Container<int, std::allocator<int> > _ci;
  // ...
};

template <typename T, typename A>
class My_Container {...};

Something<int, My_Container> sth1;
Something<int> sth2;  // use default std::vector internally
```

  - Allow to parameterize internal implementation (data structures)

Complements on Templates
Template parameters of templates (3)

- **Forbidden template parameters**
  - Local types (in C++ 2003)
  - Constants of type real
  - Non-extern pointers...

```
template <double X>  // NO
class B { ... };

const double PI = 3.41592;
B<PI> bpi; // NO
```

```
template <typename T>
class A { ... };

void f() {
  class L { ... };
  A<L> a; // NO
}
```

```
template <char *C>
class A { ... };
A<"hello"> a1; // NO

extern char c[];
  // initialize it somewhere
A<c> a2; // OK
```

Complements on Templates
Template Full Specialization

- **Full specialization of a function template**

```
template <typename T>
T Min(T a, T b) { . . . }

template <>
const char *Min<>(
     const char *s1,
     const char *s2)
{
  return strcmp(s1, s2) < 0
         ? s1 : s2;
}
```

  - The second <> is optional here (but not the first <>!)

- **Full specialization of a class template**

```
template <typename T>
class List { . . . };
```

  - a specialization of `List<T>` when `T` is a C-string

```
template <>
class List<char*> { … };
```

  - the contents and interface of `List<char*>` can be totally different from those of the generic `List<T>`

Complements on Templates
Class Template Partial Specialization

- **Partial specialization of a class template**

```
template <typename T>
class List { . . . };
```

  - a specialization of `List<T>` when `T` is a pointer

```
template <typename U>
class List<U*> { . . . };
```

  - the contents and interface of `List<U*>` can be totally different from those of the generic `List<T>`

Complements on Templates
Template specialization: rules

- **Both function and class templates can be fully specialized**
  - This includes member templates
- **Only class templates can be partially specialized**
  - Function and member templates cannot...
  - ... but they can be overloaded

```
template <typename T>
T Min(T a, T b) { . . . }
```

An overloaded form when `T` is a pointer

```
template <typename T>
T* Min(T* a, T* b) { . . . }
```

  - ~~C++11 will allow partial specialization of function templates~~
- **The specializations must be in the same namespace as the template definition**

Run Time Type Information (RTTI)
Motivation (1)

- **Make downward cast safe**

```
class A { ... };

class B : public A {
public:
 void f();      // no f() in A
};

A *pa = ...;     // may be pa points to a B?
pa->f();         // FORBIDDEN: no method f() in A
((B *)pa)->f(); // OK, but is it reasonnable?
```

Run Time Type Information (RTTI) Motivation (2)

- **C-casts are resolved at compile-time**
- **We need a way to check the cast at run-time**
  - That is to require information about the dynamic type of an object: RTTI
- **Remarks**
  - Run Time Type Information has not to be built-in, since we have already virtual functions
  - However it is more comfortable and safer to delegate its implementation to the compiler

Run Time Type Information (RTTI) Operator `dynamic_cast` (1)

- **Use with a pointer (expression) `p`**

```
T *pt = dynamic_cast<T *>(p)
```

  - Check at run-time that `p` points to an object of type `T` or (publicly) derived from `T`; otherwise return the null pointer
- **Use with a reference `r`**

```
T& rt = dynamic_cast<T&>(r);
```

  - Check at run-time that `r` is a reference to an object of type `T` or (publicly) derived from `T`; otherwise throw predefined exception `bad_cast`
- **Dynamic cast respects `const` and access control**
- **`T` must be a *polymorphic* type**

Run Time Type Information (RTTI) Operator `dynamic_cast` (2)

```
class Figure {
public:
  virtual void rotate(int);
  virtual int surface() const;
  virtual void draw() const;
};

class Ellipse : public Figure {
public:
  Ellipse(Point c, int a, int b);
  int a_axis() const;
  int b_axis() const;
};

class Circle : public Ellipse {
public:
  Circle(Point c, int r)
    : Ellipse(center, r, r) {}
  int radius() const;
};
```

```
Figure *pf1 = new Circle(c0, R);
Figure *pf2 = new Ellipse(c1, A, B);
Ellipse *pe;
Circle *pc;

x = pf1->a_axis();  // NO
pe = dynamic_cast<Ellipse *>(pf1);
if (pe != 0)
   x = pe->a_axis(); // YES... now

pc = dynamic_cast<Circle *>(pf2);
    // pc == 0

pc = dynamic_cast<Circle *>(pf1);
    // pc != 0

char *p = dynamic_cast<char*>(pf1);
   // does not compile since char is
   // not polymorphic
```


Run Time Type Information (RTTI) Class `type_info` and operator `typeid`(1)

- **Once `dynamic_cast` built into the language, the path is short to dynamic type identification**
  - A type descriptor (an instance of the predefined class `type_info`) is associated with each type (and thus with each class)
  - Operator `typeid` returns a reference to the type descriptor of its parameter

```
const type_info& typeid(type-name);
    // static resolution
const type_info& typeid(expression);
    // dynamic resolution if needed (polymorphism)
```

Run Time Type Information (RTTI)
Class `type_info` and operator `typeid`(2)

- **Class `type_info` (`#include <typeinfo>`)**

```
class type_info {
public:
  virtual ~type_info();
  bool operator==(const type_info&) const;
  bool operator!=(const type_info&) const;
  bool before(const type_info&) const; // !!
  const char *name() const;
  // No public constructor
  // No public copy operator
};
```


Run Time Type Information (RTTI)
Class `type_info` and operator `typeid`(3)

- **Operator `typeid` always returns the exact (dynamic) type of its expression**

```
Figure *pf = new Circle(...);

typeid(pf) == typeid(Figure *)    // yes
typeid(*pf) == typeid(Circle)     // yes

typeid(pf) == typeid(Ellipse *)   // no
typeid(pf) == typeid(Circle *)    // no
typeid(*pf) == typeid(Figure)     // no
typeid(*pf) == typeid(Ellipse)    // no
```


Run Time Type Information (RTTI)
Class `type_info` and operator `typeid`(5)

- **A remark on binding virtual functions within a constructor or destructor**
  - A virtual function is statically bound when called from within a base class constructor or destructor

```
class A {
public:
   virtual void f() {...}
   void g() {f();}
};
class B : public A {
public:
   virtual void f() {...}; // override
};

B b;
b.g();  // call B::f() as expected
```


Run Time Type Information (RTTI)
Class `type_info` and operator `typeid`(6)

- **`typeid` and `dynamic_cast` behave like virtual functions**
  - Thus they are statically bound when called from within a base class constructor or destructor

```
class A {public: A(); ...};
class B : public A {...};

A::A() {
  if (typeid(*this) == typeid(B)) ...
     // FALSE, even when constructing a B
  if (dynamic_cast<B *>(this) != 0) ...
     // still FALSE, even when constructing a B
}
```

ISO C++ Casts
Overview

- **Operator dynamic_cast is nice!**
  - It has a "greppable" syntax
  - It has a perfectly defined (and unique) purpose
    - Dynamically safe downward casting
- **Replacing the C casts by new casts**
  - with "greppable" syntax
  - with a unique and well-defined purpose

- **`dynamic_cast`**
  - the only cast that uses dynamic information
- **`static_cast`**
  - safe if compile-time information is enough to perform the cast
- **`reinterpret_cast`**
  - convert between pointer types or between pointers and integers
- **`const_cast`**
  - remove `const`!
  - Do not use it!

Reusing a class
Single and multiple inheritance

- **Reuse through inheritance**
  - Polymorphism and virtual functions
- **Reuse through composition**
  - Strong (value) and weak (pointer/reference) composition
- **Multiple inheritance**
  - Memory layout
  - Conversions
  - Construction
  - Virtual inheritance

Reusing a class
Reuse through inheritance

- **Static reuse mechanism**
- **Substitutability Principle**
  - The substitutability principle expresses the static semantics of (public) inheritance (a.k.a. subtyping)
  - B is a subtype of A if any instance of A may be substituted by an instance of B, for all purposes of an A

  Thus B must have at least the same features, that is the same interface as A

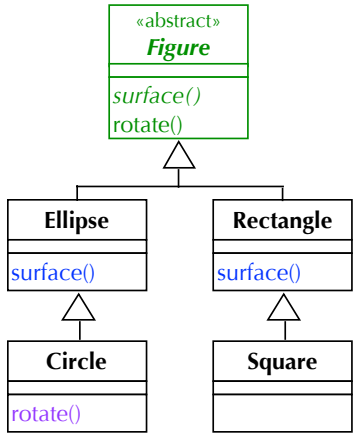
  B cannot get rid of inherited properties
- **Implementation reuse often utilizes private inheritance**
  - In this case the interfaces may be (completely) different

C++ Class: Reuse Unit
Reuse through inheritance: polymorphism(1)

```
class Figure {  // abstract class
public:
  virtual double surface() const =
0;
  virtual void rotate();
};

class Ellipse : public Figure {
public:
  double surface() const {
    return PI*a*b;
  }
};

Figure *pf = new Circle(...);
double s = pf->surface();
pf->rotate();
```



C++ Class: Reuse Unit
Reuse through inheritance: polymorphism(3)

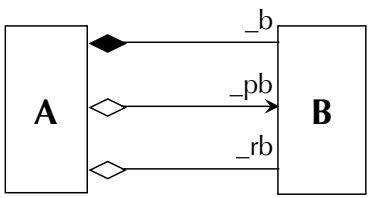
- **Any member-function can be made virtual except**
  - constructors
  - static member-functions
  - member templates (see later)
- **In particular, the destructor can be virtual**
  - and should be for abstract classes
- **Friend functions are not members; they thus cannot be virtual**
  - at least in the class they are friend of
  - if they are members of some other class, they can be virtual in this other class

C++ Class: Reuse Unit
Reuse through composition (1)

- **Composing with an instance (composition by value)**
  - Static composition
  - Different composites cannot share the component
  - The composite is "responsible" for its components, it "owns" them
  - The lifetimes of the composite and its components are identical
- **Composition with a pointer or a reference**
  - Possibly, dynamic composition
  - Make polymorphism possible when delegating
  - Sharing is possible
  - Objects (component and composite) may be independent

C++ Class: Reuse Unit
Reuse through composition (2)

```
class B { };
class A {
  B _b;  // by value
  B* _pb;// by pointer
  B& _rb;// by reference
public:
  A(const B& b1, B& b2)
    : _b(b1),
      _pb(&b2),_rb(b2)
  {}
};
```

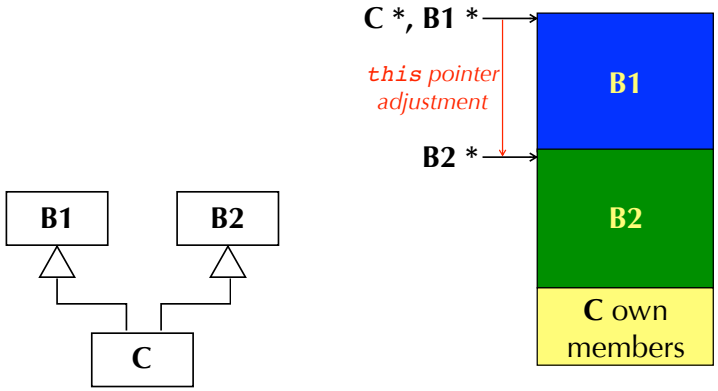


- During `A` construction, `b1` is copied into `_b` (by `B` copy constructor: value semantics)
- During `A` destruction, the destructor of `B` will be invoked

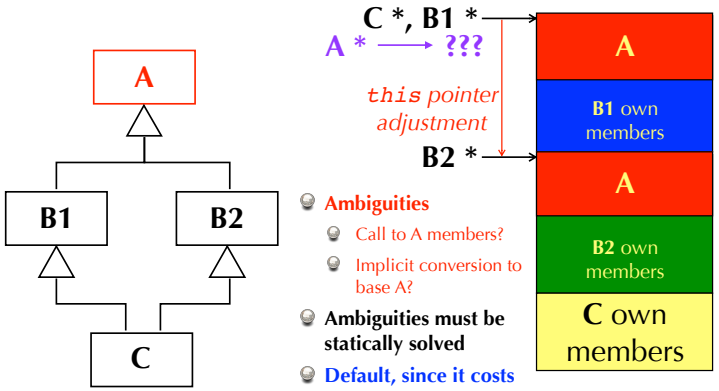
Multiple inheritance
Reminders

- **The same class cannot appear several times as a direct base class**
- **The inheritance graph must be acyclic (DAG)**
- **Ambiguities are detected at compile time and must be statically resolved**
  - Only effective ambiguities are detected

Multiple inheritance
Regular (non virtual) multiple inheritance (1)

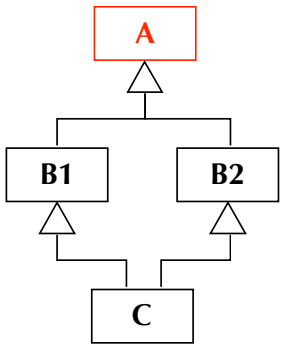


Multiple inheritance
Regular (non virtual) multiple inheritance (2)



- **Ambiguities**
  - Call to A members?
  - Implicit conversion to base A?
- **Ambiguities must be statically solved**
- **Default, since it costs nothing!**

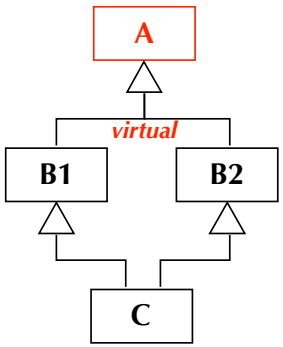
Multiple inheritance
Regular (non virtual) multiple inheritance (3)




```
class A {
public:
  void f() {}
};
class B1 : public A {};
class B2 : public A {};
class C : public B1, public B2 {};

C c;
c.f(); // ambiguous!
static_cast<B1&>(c).f(); // OK
       // or override f() in C
A *pa = static_cast<A *>(&c);
       // ambiguous!
A *pa = static_cast<A*>(
           static_cast<B1 *>(&c));
```

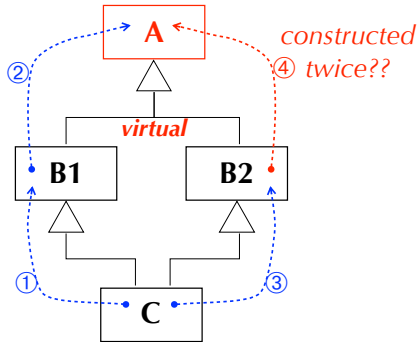

Multiple inheritance
Virtual multiple inheritance (1)




```
class A {
public:
  void f() {}
};
class B1 : virtual public A
{};
class B2 : virtual public A
{};
class C : public B1, public B2
{};

C c;
c.f(); // no longer ambiguous!
A *pa = static_cast<A *>(&c);
       // no longer ambiguous!
```

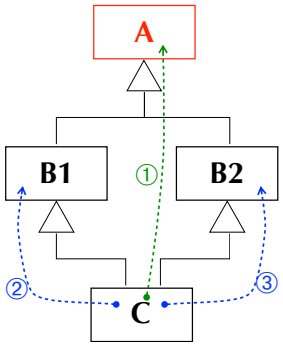
Multiple inheritance
Virtual multiple inheritance (2)



- **Construction of a virtual base class: *regular construction?***

```
struct A {
  A() {}
};
struct B1 : virtual A {
  B1() : A() {}
};
struct B2 : virtual A {
  B2() : A() {}
};
struct C : B1, B2 {
  C() : B1(), B2() {}
};
```


Multiple inheritance
Virtual multiple inheritance (3)



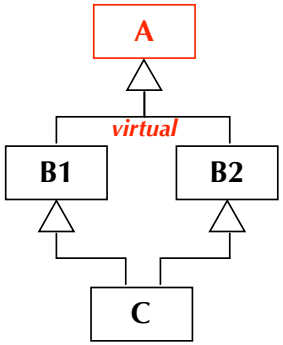
- **Construction of a virtual base class: virtual base first**

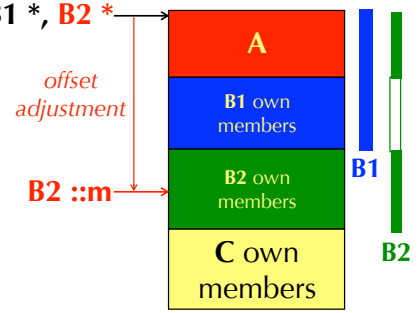
```
struct A {
  A() {}
};
struct B1 : virtual A {
  B1() : A() {}
};
struct B2 : virtual A {
  B2() : A() {}
};
struct C : B1, B2 {
  C() :      B1(), B2()
{}    A(),
};
```


Multiple inheritance
Virtual multiple inheritance (5)

- **Virtual base classes must be constructed exactly once (as any C++ object)**
- **They are constructed before any "regular" base class**
- **Thus they must**
  - Either be constructible by default
  - Or be (explicitly) constructed by any (concrete) derivative, whatever how deep this derivative is in the hierarchy

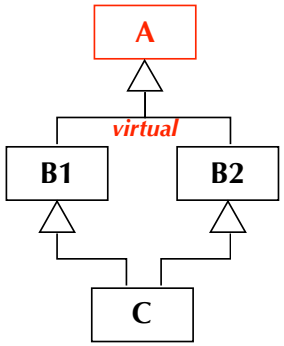
Multiple inheritance
Virtual multiple inheritance (6)

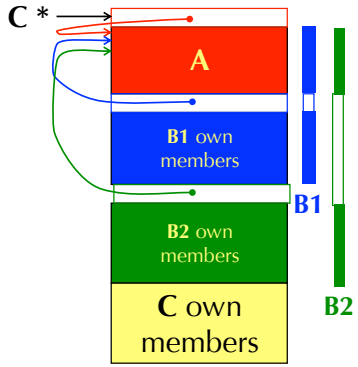




Not the real layout

Multiple inheritance
Virtual multiple inheritance (7)

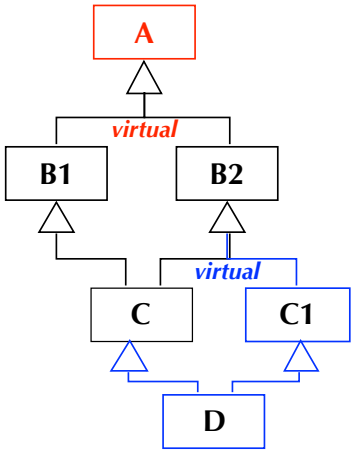




Real layout

- Symmetry B1/B2
- Facilitate conversions, virtual calls

Multiple inheritance
Virtual multiple inheritance (7)



- **Multiple virtual base classes?**

Multiple inheritance
Virtual multiple inheritance (8)

- **The complexity of conversions and addressing explains why virtual derivation is not the default in C++**
- **Note that pointers to members take into account this complexity**
- **Operator `dynamic_cast` makes downward cast possible for virtual base classes**
  - `static_cast` does not give a correct result in case of virtual inheritance
- **This is not so bad as it seems!**
  - C++ has no (mandatory) `Object` class nor single inheritance tree
  - More likely a lot of small trees (a mediterranean forest?)
  - Templates and duck typing spare inheritance hierarchies

Name Spaces

- **Class scope**
- **Name lookup**
- **Name packaging (`namespace`)**

Name Spaces
Class scope (1)

```
class List {
private:
  static int nb_lists;  // List::nb_lists
  enum {N1, N2};        // List::N1, List::N2
  class Cell {          // List::Cell
    int info;
    Cell *next;
     // ...
  };
  Cell *first;          // this->first
  Cell *last;           // this->last
public:
  static void fst();    // List::fst()
  void method();        // this->List::method()
  // ...
};
```


Name Spaces
Class scope (2)

- **Definition of class scope**
  - The body of the class definition
  - The body of the member-functions from the `::` operator of the function name
  - Friend functions are not within the class scope

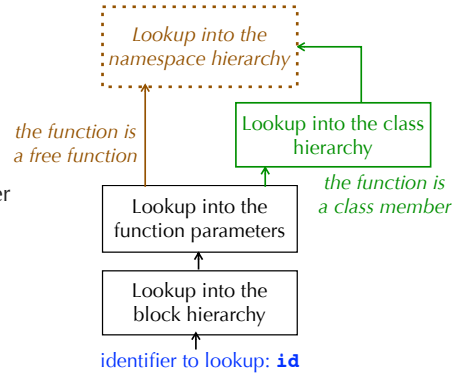
```
class A {
  class B { };
  B m(B);
  friend void f(B);
  // ...
};

A::B A::m(B b) {
  B b1;
  // ...
}

void f(A::B b) {
  A::B b2;
  // ...
}
```


Name Spaces: name lookup
Hierarchical name space with inheritance (1)

- **Name lookup**
  - This is the simple case
  - See also namespaces further



Name packaging (`namespace`)
Motivation

- **Avoid name collisions due to an unique global name space**
  - when merging programs
  - when using several libraries...
- **Classes were already name spaces**
  - Explicit qualification (operator `::`) is required outside the class
  - The name space of a class cannot be extended
    - Inheritance creates a new (nested) name space

Name packaging (namespace) Basics

- **Definition**

```
namespace My_Lib {
  class List {...};
  void f(int, double);
  int f(char);
  // …
  namespace Sub_Lib {
    class Map {…};
    // …
  }
}
```

  - Namespaces can be nested
  - A namespace is "open" : it can be extended simply by opening a new namespace clause with the same name

- **Usage**
  - Explicit qualification

```
My_Lib::List l;
My_Lib::f(3, 12.5);
int i = My_Lib::f('a');
My_Lib::Sub_Lib::Map m;
```

  - Using declaration

```
using My_Lib::f;
f(3, 12.5);
int i = f('a');
```

  - Using directive

```
using namespace My_Lib;
List l;
f(3, 12.5);
int i = f('a');
Sub_Lib::Map m1;
```


Name packaging (namespace) Backward compatibility (1)

- **Standard ISO C++ library are in a predefined name space (`std`)**
- **Standard ANSI C header files are also part of ISO C++, but their name is prefixed with `c` and the extension `.h` is dropped**
  - e.g., `<cstdio>`, `<cstring>`, `<cctypes>`...
  - All standard ANSI C functions are part of the `std` namespace (`prinft`, `sin`...)
- **The C++ standard header files**
  - do not have a `.h` or `.hpp` extensions
  - do not contain any `using` directive (`using namespace std`)

Name packaging (namespace) The global namespace

- **Unary operator `::` accesses the global namespace**

```
string aGlobalString; // global variable
void f() {
  cout << ::aGlobalString << endl;
}
```

  - It corresponds to the C global namespace
- **The global namespace contains**
  - the names of the top level namespaces
  - the global names that are not in a specific namespace
- **It must be collision free**
- **It is not to be confused with the standard namespace `std`**

Name packaging (namespace) Anonymous (unnamed) namespaces

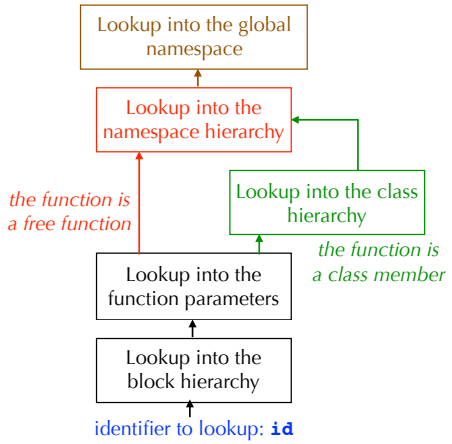
- **Unnamed namespace**

```
namespace {
  int x;
  void f(int);
  class A { … };
  typedef std::string My_Tag;
}
```

- **Its elements are directly accessible in the current compilation unit, but are not accessible outside**
- **Equivalent to `static` global in C (file scope)**
  - but more general since can contain **types** (`A`, `My_Tag`)
  - Hence static global deprecated in C++

Name packaging (namespace) Namespaces and name lookup (1)

- **Regular lookup**
  - Namespaces form just another nesting hierarchy
  - The using directives and declarations import the corresponding names into another namespace



Name packaging (namespace) Namespaces and name lookup (2)

- **Problem when looking up unqualified names**
  - Regular lookup is not able to find which f is used
  - Here, we can of course use full qualification…
  - But it is not always suitable

```
namespace X {
  class A { ... };
  void f(A a);
  std::ostream&
    operator<<(std::ostream& os, A a);
}

int main() {
  X::A a;
  f(a);
  X::f(a);

  std::cout << a;
  X::operator<<(std::cout, a);
     // not very nice, is it?
}
```


Name packaging (namespace) Namespaces and name lookup (3)

- **Agument dependent lookup (ADL) (cont.)**
  - A.k.a. Koenig's lookup
  - When resolving an unqualified name in a (non-member) function call, look into the namespaces and classes associated with the arguments
  - The candidate functions will be the union of regular look up and ADL ones

```
namespace X {
  class A { ... };
  void f(A a);
  std::ostream&
    operator<<(std::ostream& os, A a);
}

int main() {
  X::A a;
  f(a); // ADL finds X::f

  std::cout << a;
        // ADL finds X::operator<<

  std::string s;
  std::cout << s << std::endl;
        // ADL finds std::operator<<
        // for string and endl
}
```


Name packaging (namespace) Namespaces and name lookup (4)

- **Argument dependent lookup (ADL) (cont.)**
  - ADL behaves as if the name had been prefixed by an associated namespace
  - Except that all using directives are ignored in the associated namespaces

```
namespace X {
  template <typename T>
  void f(T t) { ... }
}

namespace Y {
  using namespace X;
  class A { ... };
}

int main() {
  Y::A a;
  f(a); // f not found
  Y::f(a); // f found by regular
           // lookup
}
```

Name packaging (namespace) Namespaces and name lookup (5)

- **Agument dependent lookup (ADL) (cont.)**
  - Since the non-dependent names are looked up in the union of regular look up and ADL, there are possible ambiguities

```
// <utility> has a min() template
#include <utility>
using namespace std;

// another min() template
template <typename T>
const T& min(const T&, const T&);

int main() {
  std::string s1, s2;

  std::string s = min(s1, s2);
    // Regular lookup finds ::min
    // ADL finds std::min
    // Both are exact matches
    // Thus the call is ambiguous
}
```


Name packaging (namespace) Namespaces and name lookup (6)

- **Name hiding in nested namespaces**
  - Redefining a name in a nested namespace hides the outermost ones
    - and does not overload them
  - A hidden name is not considered during name lookup (regular or ADL)

```
namespace X {
  void f(int);

  namespace Y {

    void f(double);
      // this f hides the one above
    void g() { f(3);}
      // always call Y::f(double)
  }
}

int main() {
  X::Y::g(); // calls X::Y::f
  using namespace X;
  using namespace X::Y;
  f(3); // calls X::f
  g();  // still calls X::Y::f
}
```


Name packaging (namespace) Namespaces and name lookup (7)

- **Name hiding in nested namespaces (cont.)**
  - A using declaration imports a name, thus allowing to overload it
  - The same remark also applies to inheritance class hierarchies

```
namespace X {
  void f(int);

  namespace Y {
    using X::f;
    void f(double);
      // now f is overloaded
    void g() { f(3);}
      // calls X::f(int)
  }
}

int main() {
  X::Y::g();
  using namespace X::Y;
  f(3);   // calls X::f
  f(3.5); // calls X::Y::f
  g();    // calls X::f
}
```


Name packaging (namespace) Classes as namespaces (1)

- **Classes are namespaces**
- **Directive `using namespace` is forbidden for a class**
- **But using declaration is possible**
  - it allows to import an inherited member-function to the local class scope
  - or to adjust the access control level in the derived class of an inherited member

Name packaging (namespace) Classes as namespaces (2)

- **Importing inherited members in the global scope**

```
class A {
public:
  void f(char);
  void g(double);
};
class B : public A {
public:

  void f(int);  // hiding inherited f(char)
};

B b;
b.f('a'); // compilation warning: calls f(int)!
```


Name packaging (namespace) Classes as namespaces (3)

- **Importing inherited members in the derived class scope**

```
class A {
public:
  void f(char);
  void g(double);
};
class B : public A {
public:
  using A::f;
  void f(int);  // overloading inherited f(char)
};

B b;
b.f('a'); // calls f(char)
```


Name packaging (namespace) Classes as namespaces (4)

- **Adjusting access control level of inherited members**

```
class A {
public:
  void f(char);
  void g(double);
};
class C : private A {
public:

};

C b;
c.f('a'); // f(char) is not accessible
```


Name packaging (namespace) Classes as namespaces (5)

- **Adjusting access control level of inherited members**

```
class A {
public:
  void f(char);
  void g(double);
};
class C : private A {
public:
  using A::f;
};

c b;
c.f('a'); // f(char) is now also public in C
```

Advanced C++ Libraries

Part 4

The Standard Template Library (STL)

Advanced C++
The Standard Template Library (STL)

- **Contents**
  - General purpose utilities
  - Containers, iterators, and algorithms
  - Character strings
  - Input-output streams
  - (Numerics)
  - (Localization)
- **Implementation constraints**
  - Usable directly and easily
  - Efficient
  - Primitive
  - Complete
  - Extensible
  - Neutral policy
    - Compatible with base types
    - Support for all programming styles
  - Type-safe
    - No cast needed to use it

The Standard Template Library
Summary

- **Organization of the STL**
- **Some utilities**
- **Containers, iterators, and algorithms**
- **High order programming**
- **Algorithms**
- **Character strings**
- **Input-output streams**
- **The STL and exceptions**
- **Header files**

Overview of the STL
Organization

- **Namespace `std`**
- **Header files**
  - Recover ANSI C header files (`<cXXXX>` ≡ `<XXXX.h>`)
- **Library elements**
  - Containers, iterators, algorithms
  - Character strings, input-output streams
  - General utilities, diagnostics
  - Basic run-time support
  - (Localization)
  - (Numerical elements)

Utilities
Class `pair`

- **Couple of values**

```
template <typename T1, typename T2>
struct pair {
  T1 first;
  T2 second;
  // Constructors
  pair();
  pair(const T1&, const T2&);
  template <typename U1, typename U2>
  pair(const pair<U1, U2>& p);
};
```

- **Other operations**
  - Equality, relational operators
- **Convenience function template `make_pair`**

```
template <typename T1, typename T2>
pair<T1, T2> make_pair(const T1&, const T2&);
```


Utilities
Various utilities

- **Numeric limits: class `numeric_limits<T>`**
- **Minimum and maximum functions: `min()` and `max()`**

```
template <typename T>
const T& min(const T&, const T&);
```

  - Possibility to pass the comparison predicate as third parameter

```
template <typename T, typename Compare>
const T& min(const T&, const T&, Compare comp);
```

    - `comp(a, b)` should return true if `a` is less than `b` for the given order relationship
- **Swap values: `swap()`**

```
template <typename T>
void swap(T&, T&);
```


Containers and Iterators

- **Container "by value"**
  - Elements are copied into the container
  - Use containers of *pointers* if you do not want to copy
- **Homogeneous contents**
  - Yet, different implementations, capabilities, and performances
- **Homogeneous interface**
  - Common member functions
  - Common iterator interface
- **Specific member functions**
- **Parametrized memory allocation scheme (template parameter)**

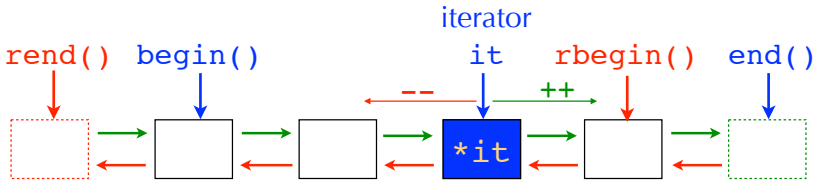
```
template <typename T,
          typename Allocator = allocator<T> >
class list {...};
```


Containers and Iterators
Main containers of STL 03

Basic containers (a.k.a. Sequences)		
`vector<T>`	`<vector>`	Array with automatic (re)allocation
`deque<T>`	`<deque>`	Sequence with optimized indexing
`list<T>`	`<list>`	Doubly linked sequence (list)
Container adapters: Default container is `deque` (for `priority_queue`, it is `vector`)		
`stack<T>`	`<stack>`	Last In First Out – no iterator
`queue<T>`	`<queue>`	First In First Out – no iterator
`priority_queue<T>`	`<queue>`	Queue with priority (`operator<()` – no iterator
Associative containers		
`map<K, T>`	`<map>`	Associative array without duplicate (`K` is the key)
`multimap<K, T>`	`<map>`	Associative array with duplicate (`K` is the key)
`set<K>`	`<set>`	Set (without duplicates) of `K`
`multiset<K>`	`<set>`	Set (with duplicates) of `K` (a.k.a. *Bag*)

Containers and Iterators
Iterators (1)

- **Model : sequence and extended pointer**




```
list<int> L;
for (int i = 0; i < N; i++) L.push_front(i);
list<int>::const_iterator it;
for (it = L.begin(); it != L.end(); ++it)
   cout << *it << ' ';
```


Containers and Iterators
Iterators (2)

- **Reverse iteration**
  - Use a reverse iterator

```
list<int>::const_reverse_iterator rit;
for (rit = L.rbegin(); rit != L.rend(); ++rit)
   cout << *rit << ' ';
```

- **Non const iterator**
  - Allow modification of container elements

```
list<int>::reverse_iterator rit;
for (rit = L.rbegin(); rit != L.rend(); rit++)
   *rit = 10;
```


Containers and Iterators
Iterators (3)

- **Using iterators as interval bounds (range)**

```
list<int> L;
list<int>::iterator it;
int n;
// ...
for (n = 0, it = L.begin();
   (it = find(it, L.end(), 3)) != L.end(); ++it)
{
  n++;
}
```

  - Iterator ranges always include left bound and exclude right one ([… [)

Containers and Iterators
Models of iterators

- **Input iterator**
  - Move forward only (++)
  - Provide read only access
  - Cannot process the same element several times
- **Output iterator**
  - Move forward only
  - Provide read/write access
  - Do not have comparison operators
- **Forward iterator**
  - Both input and output
  - Have comparison operators
- **Bidirectional iterator**
  - Move both ways (++, --)
- **Random access iterators**
  - Bidirectional
  - Have arithmetic and comparison operators
- **The model of iterator depends on the type of the container**
- **Member functions and algorithms specify which model they need**

Containers and Iterators
Pure abstraction and iterators

- **The STL mainly relies on the principle of Pure Abstraction**
  - Also known as Duck Typing
    - *"If it walks like a duck and quacks like a duck, it's a duck"*
  - This principle spares inheritance hierarchies…
  - It replaces inheritance with type conformance
- **Iterators are smart pointers, they look like pointers…**
- **But pointers look like iterators, hence they are iterators**

```
int tab[10];
int *pi = find(&tab[0], &tab[10], 5);
if (pi != &tab[10]) ...
```


Containers and Iterators
Common members

- **Member types**
- **Constructors**
  - default, copy, from iterator range
- **Destructor, `clear()`**
- **Iterator interface (for iterable containers)**
- **Relational operators**
  - if value type has them
- **Copy assignment**
  - if value type has it
- **`swap()`**
- **`size()`**
- **Element insertion and removal…**

Containers and Iterators
Specific members: sequences

- **All sequences**
  - Operations on head and tail: insertion, consultation
- **Vectors and double ended queues**
  - Preallocation (`reserve()`, `capacity()`)
  - Indexing:
    - `at(i)` : throws `std::out_of_range` if index out of bounds
    - `operator[]` : does not check nor throw
- **Lists**
  - Remove duplicates (`unique()`), `sort()`, `merge()`
  - `splice()`: insert a sublist into another list

Containers and Iterators
Specific members: associative containers (1)

- **Maps and multimaps**
  - Value type is `pair<const K, T>`

```
map<string, int> m;
// ...
it = m.find("hello");
       // *it is a pair<const string, int>
if (it != m.end())
   cout << it->first << ' ' << it->second;
it->first = "bye";   // NO!
```

Containers and Iterators
Specific members: associative containers (2)

- **Maps only**
  - Associative array using `T& operator[](const K&)`
  - This operator allocates a node for the key `k` if not already in the map
  - The second element is initialized to the zero of its type (`T`)

```
map<string, int> m; // initially empty
m["hello"] = 3;     // new element ("hello", 3)
cout << m["hello"]; // print 3
cout << m["bye"];   // allocate new element
                    // ("bye", 0) and print 0
```


Containers and Iterators
Remarks

- **Constraints on the elements (`value_type`)**
  - Default constructor
    - Define the "zero" of the type
  - Copy operations
  - For ordered containers, "less than" operator (`<`)
    - If no equality operator, "less than" is used to generate one:
      `a == b ≡ !(a < b) && !(b < a)`
- **Vectors**
  - No arithmetics (see `valarray`)
  - Specialization `vector<bool>` is defined
- **Sets**
  - No set operations (union, intersection…)
    - Use algorithms on sorted collections

Containers and Iterators
Operation complexity

	Indexing	Insertion anywhere	Operation on head	Operation on tail
vector	O(1)	O(n)+		O(1)+
list		O(1)	O(1)	O(1)
deque	O(1)	O(n)	O(1)	O(1)
stack		O(n)		O(1)
queue		O(n)	O(1)	O(1)
priority_queue		O(log n)	O(log n)	
map	O(log n)+	O(log n)+		
multimap		O(log n)+		
set		O(log n)+		
multiset		O(log n)+		
string	O(1)	O(n)+	O(n)+	O(1)+
array	O(1)			
valarray	O(1)			
bitset	O(1)			

Algorithms

- **About sixty predefined algorithms**
  - Non-modifying algorithms
  - Modifying algorithms
    - Simple modifying algorithms
    - Removing algorithms
    - Mutating algorithms
    - Sort and merge algorithms
  - Algorithms on sorted collections
  - Numeric algorithms
- **Header file `<algorithm>`**

Algorithms
Algorithms on sequences (1)

- **Use iterators**
  - to specify their range of action
  - to retrieve the result

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 7);
```

- **Have other parameters indicating an action to perform on each element of the sequence**
  - either selecting elements
  - or modifying elements

```
list<int> L;
list<int>::iterator it = find(L.begin(), L.end(), 7);
```


Algorithms
Algorithms on sequences (2)

- **Filtering a sequence**

```
bool is_gt_4(int i) {
  return i > 4;
}

list<int> L;
int n = count_if(L.begin(), L.end(), is_gt_4);
```

- **Transforming a sequence**

```
int len(string s) {
  return s.size();
}

list<string> LS(10, "hello");
vector<int> VI(LS.size());   // preallocate

transform(LS.begin(), LS.end(), VI.begin(), len);
```


Algorithms
Function-objects (1)

- **Algorithms `for_each`, `find_if`, `count_if`... are function templates**

```
template <typename InputIterator,
          typename UnaryPredicate>
InputIterator find_if(InputIterator first,
                      InputIterator last,
                      UnaryPredicate pred)
{
  InputIterator it;
  for (it = first; it != last && !pred(*it); ++it)
  {}
  return it;
}
```


Algorithms
Function-objects (2)

- **`UnaryPredicate` denotes a type that may be**
  - either a pointer to a function (such as `is_gt_4`)
  - or a class defining `operator()`
    - A class of "function-objects", a *functor*
- **In all cases the signature of the "function" should be something like**

```
bool pred(InputIterator::value_type)
```

Algorithms on Sequences/Sets
Function-objects (3)

- **A function-object is an instance of a class defining `operator()`**

```
class is_gt_4 {
public:
  bool operator()(int i) {return i > 4;}
};

is_gt_4 criter; // define an object
if (criter(4)) ... // criter.operator()(4)

list<int> L;
int n = count_if(L.begin(), L.end(), criter);

int n = count_if(L.begin(), L.end(), is_gt_4());
            // call is_gt_4  default constructor
```


Algorithms on Sequences/Sets
Function-objects (4)

- **What are the advantages of function objects?**
  - They can have attributes

```
class sum_gt_n {
  int _n;
  int _sum;
public:
  sum_gt_n(int n) : _n(n), _sum(0) {}
  void operator()(int i) {
     if (i > _n) _sum += i;
  }
  int sum() const {return _sum;}
};
int s = for_each(L.begin(), L.end(), sum_gt_n(5)).sum();
```

  - The attributes can be used as function extra parameters
  - They can accumulate information while traversing the sequence

Algorithms on Sequences/Sets
Function-objects (5)

- **Using operators in algorithms?**
  - Trying to negate a list of integers

```
list<int> l1(…);
list<int> l2(l1.size());
transform(l1.begin(), l1.end(), l2.begin(), operator-);
```

  - This does not compile
    - The compiler has not enough information to deduce the type of the operator
  - Encapsulating operators in functors

```
template <typename T> struct negate {
  T operator()(T t) {return –t;}
};
transform(l1.begin(), l1.end(), l2.begin(), negate<int>());
```


Algorithms on Sequences/Sets
Function-objects (6)

- **Predefined STL function-objects for operators**

```
negate<type>(p)              -p
plus<type>(p1, p2)           p1 + p2
   and minus, multiplies,
   divides, modulus
equal_to<type>(p1, p2)       p1 == p2
   and not_equal_to
less<type>(p1, p2)           p1 < p2
   and greater, less_equal,
   greater_equal
logical_not<type>(p)         !p
logical_and<type>(p1, p2)    p1 && p2
   and logical_or
```

Algorithms on Sequences/Sets
Functionals (1)

```
class Figure {
public:
  void draw() const;
  double rotate(double angle);
  // ...
};

list<Figure> LF;
for_each(LF.begin(), LF.end(), &Figure::draw);
                  // Error !
```

- Algorithm `for_each` expects a regular function `f(Figure)`
- `&Figure::draw` is a pointer to a member function

Algorithms on Sequences/Sets
Functionals (2)

```
class Figure {
public:
  void draw() const;
  double rotate(double angle);
  // ...
};

list<Figure> LF;
for_each(LF.begin(), LF.end(),
         mem_fun_ref(&Figure::draw));
```

- `mem_fun_ref` is an example of functional
- It transforms a pointer to member into a free function

High Order Programming
Boost and TR1 `bind` adaptor (1)

- **Drawbacks of STL (2003) function adaptors**
  - Distinction between different natures of functions
    - regular functions
    - member functions
    - well-formed function objects
  - Distinction between the value type of containers (pointers or values)
  - Functions must have no more than 2 parameters
  - Difficult to compose functions
  - Terrible syntax

```
bind2nd(mem_fun(&Figure::rotate), 3.141592));
```

  - Possible problems with parameters passed by reference…
  - Problem with containers of smart pointers

Algorithms on Sequences/Sets
Boost and TR1 `bind` adaptor (2)

- **The `bind` function object adapter**
  - Part of TR1, in `std` since C++11

```
#include <tr1/functional> // C++03
using namespace std::tr1::placeholders;

#include <functional>     // C++11
using namespace std::placeholders;
```

  - Allow to adjust the parameter list of any callable to make it suitable for use in an algorithm
- **Notion of a "callable"**
  - Anything which looks like a function for duck typing
    - a regular (free) function
    - a function object
    - a member function or a pointer to a member function

High Order Programming
Boost and TR1 `bind` adaptor (3)

`bind<Return>(f, p1, p2, …, pn)`

- `<Return>` is optional if the compiler can deduce it

if present, `Return` designates the return type of the function object

- `f` may be a (pointer to) a regular function or a function object with `n` parameters
- `f` may be a (pointer to an) instance member function with `n-1` parameters (not accounting for this which will become the first parameter of the function object)
- The parameters `pi` may be
  - either an expression which will be bound to the corresponding parameters of `f`
  - or a placeholder such as `_1`, `_2`, `_3`… which corresponds to a free variable, thus a parameter of the produced function object
- The function object produced will have
  - as many parameters as the number of different placeholders used (they must be used in order)
  - in C++03, its parameters are passed *by reference* to a variable (thus the effective parameter cannot be a literal value nor an expression yielding a value)

High Order Programming
Boost and TR1 `bind` adaptor (4)

Examples of function objects (φ) created by `bind`

```
int f(int a, int b) { return a - b; }

bind(f, _1, 3)    ➔   φ(x)    = f(x, 3)
bind(f, 4, 3)     ➔   φ()     = f(4, 3)
bind(f, _1, _2)   ➔   φ(x, y) = f(x, y)
bind(f, _1, _1)   ➔   φ(x)    = f(x, x)
bind(f, _2, _1)   ➔   φ(x, y) = f(y, x)

int n = 4, m = 3;
bind(f, _1, 3)(n)      == 1
bind(f, _1, _2)(n, m)  == 1
bind(f, _1, _1)(n)     == 0
bind(f, _2, _1)(n, m)  == -1
bind(f, _1, _1)(3)     ➔ does not compile in C++03
bind(f, _1, _1)(n + 1) ➔ does not compile in C++03
```


High Order Programming
Boost and TR1 `bind` adaptor (5)

Usage of `bind`

```
struct Figure {
  void draw() const;
  double rotate(double angle);
  double surface() const;
};

dequeue<Figure *> dq;
// . . .
for_each(dq.begin(), dq.end(),
         bind(&Figure::draw, _1));
list<double> li;
transform(dq.begin(), dq.end(), back_inserter(li),
          bind(&Figure::rotate, _1, 3.141592));
```


High Order Programming
Boost and TR1 `bind` adaptor (6)

Function composition with `bind`

```
struct Figure {
  void draw() const;
  double rotate(double angle);
  double surface() const;
};

dequeue<Figure *> dq;
// . . .
sort(dq.begin(), dq.end(),
   bind(less<double>(),
        bind(&Figure::surface, _1),
        bind(&Figure::surface, _2)));
```

High Order Programming
Boost and TR1 `bind` adaptor (7)

- **Binding data members**

```
map<int, string> m;
typedef map<int, string>::value_type pair_type;
void print_string(const string& s);
// . . .
for_each(m.begin(), m.end(),
         bind(&print_string,
              bind<string>(&pair_type::second,
_1)));
```

  - Binding a data member is like an accessor

High Order Programming
Boost and TR1 `bind` adaptor (8)

- **Parameters of `bind`**
  - The parameters of a `bind` expression are ***copied*** into the function object
  - If not desired, use *reference wrappers* `ref` and `cref`

```
string add_suffix(const string& s, string& suffix) {
     suffix += "-";
     return s + suffix;
}
list<string> ls;
string suffix = "";
// . . .
transform(ls.begin(), ls.end(), ls.begin(),
          bind(&add_suffix, _1, ref(suffix)));
```

  - `ref(v)` encapsulates a reference to a variable
    - `ref` objects are ***copiable***, whereas ordinary references are not
  - If a reference to a constant is needed, uses `cref(v)`

High Order Programming
Boost and TR1 `bind` adaptor (9)

- **Advantages of `bind`**
  - Function objects with up to 9 parameters
  - Possibility of argument reordering
  - Compatible with smart pointers such as `shared_ptr` (`mem_fun` is not)
  - Compatible with reference wrappers (`ref` and `cref`)
  - Possibility to bind data members (a sort of getter)
  - Possibility of function object composition
- **Beware!**
  - In TR1 (before C++11), the placeholders are in `std::tr1::placeholders`
    - In C++11, they will be in `std::placeholders`
    - In Boost they are in `::`, the anonymous namespace
  - `bind` is not compatible with STL 03 binders (`bind1st`, `bind2nd`, etc.)

Algorithms
Non-modifying algorithms (1)

- **Modify neither the collection itself nor the value of the elements**
- **Require associated operations on the elements (equality, comparison...)**

`for_each()`	Perform an operation on each element
`count()` `count_if()`	Count number of elements with some value (satisfying a condition)
`min_element()` `max_element()`	Return the smallest (largest) element
`equal()`	Return whether two collections are equal
`mismatch()`	Return the first element that differs in two sequences
`lexicographical_compare()`	Lexicographic comparison of two collections

Algorithms
Non-modifying algorithms (2)

`find()` `find_if()`	Find the first element with some value (satisfying a condition)
`search_n()`	Search n consecutive elements satisfying a condition
`search()` `find_end()`	Search for first (last) occurrence of a subrange
`find_first_of()`	Search for the first of several elements
`adjacent_find()`	Search for two consecutive elements that are equal

Algorithms
Modifying algorithms (1)

- **Simple modifying algorithms**
  - Modify the elements
  - Create new collections
- **Removing algorithms**
  - "Remove" elements from a collection
    - Not a real removal (see further)
- **Mutating algorithms**
  - Change the order of elements
- **Sorting algorithms**
  - Special form of mutating algorithms
  - Sort a collection
- **Sorted range algorithms**
  - Only operate on sorted collections
- **Numeric algorithms**
  - Apply "numeric" operations on collections

Algorithms
Modifying algorithms (2)

- **The modifying algorithms *never* modify the number of elements in the container**
  - They may modify the order of the elements
  - And also the value of the elements themselves

```
deque<double> dq(10, 3.14); // 10 elements
list<double> ld;  // empty by default
copy(dq.begin(), dq.end(), ld.begin());
            // likely to crash!!
```


Algorithms
Modifying algorithms (3)

- **Insert iterators**
  - Special iterators for which ++ (and --) do allocate new elements
    - `inserter`, `back_inserter`, `front_inserter`

```
deque<double> dq(10, 3.14); // 10 elements
list<double> ld;  // empty by default
copy(dq.begin(), dq.end(), back_inserter(ld));
      // new elements added at the end of ld
```

Algorithms
Simple modifying algorithms

`for_each()`	Perform an operation on each element
`copy()` `copy_backward()`	Copy a collection starting with first (last) element
`transform()`	Transform a collection into another collection or combine elements of two collections to form a third one
`merge()`	Merge two sorted collections
`swap_range()`	Swap elements of two collections
`fill()` `fill_n()`	Replace each (n) element(s) with a given value
`generate()` `generate_n()`	Replace each (n) element(s) with the result of an operation
`replace()` `replace_if()`	Replace elements with a given value (that match a condition) with another value
`replace_copy()` `replace_copy_if()`	Same as above, except that the result goes into another collection

Algorithms
Removing algorithms (1)

`remove()` `remove_if()`	Remove elements with a given value (satisfying a condition). In fact reorganize a collection so that it contains the elements to keep in an initial subcollection. Return the end of this initial subcollection
`remove_copy()` `remove_copy_if()`	Copy elements that do not match a given value (do not satisfy a condition) in another collection
`unique()`	Remove adjacent duplicates (works the same way as `remove()`)
`unique_copy()`	Copy elements in another collection while removing adjacent duplicates

- **Associative containers do not support `remove()` nor `unique()`**

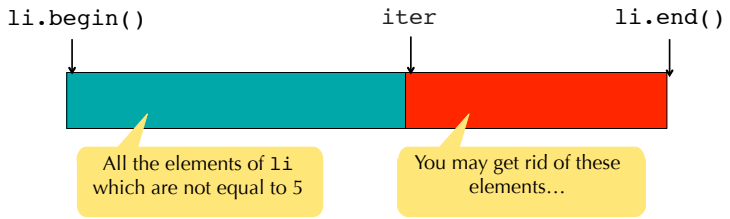
Algorithms
Removing algorithms (2)

- **`remove` and `remove_if` do not remove anything!**

```
list<int> li(…);
iter = remove(l.begin(), l.end(), 5);
```

- Hence the *idiom*

```
li.erase(remove(li.begin(), li.end(), 5), li.end());
```



Algorithms
Mutating algorithms

`reverse()` `reverse_copy()`	Reverse the order of elements in a collection (copying into another collection)
`rotate()` `rotate_copy()`	Circular permutation of the elements in a collection (copying into another collection)
`next_permutation()` `prev_permutation()`	Permutate the order of elements
`random_shuffle()`	Bring the elements in a random order
`partition()` `stable_partition()`	Change the order of elements so that the ones matching a condition are at the front (preserving the relative order of the original collection)

- **Associative containers do not support these operations**

Algorithms
Sorting algorithms (1)

- **These algorithms sort the elements according to a sort criterion**
  - `operator<` by default
  - or a binary predicate given as a (last) parameter of the algorithm
- **Beware: `sort()` and `stable_sort()` cannot be used for `list`**
  - They require a random access iterator
  - Use the `list::sort()` member function instead

Algorithms
Sorting algorithms (2)

`sort()` `stable_sort()`	Sort a collection (preserving order of equal elements)
`partial_sort()` `partial_sort_copy()`	Sort until an initial subcollection is correctly sorted (copying into another collection)
`nth_element()`	Sort a collection yielding two subcollections separated by a given element

`make_heap()`
`push_heap()`
`pop_heap()`
`sort_heap()`
	Heap manipulation (heaps are a special representation of binary trees used in sorting algorithms)

`partition()` `stable_partition()`	Change the order of elements so that the ones matching a condition are at the front (preserving the relative order of the original collection)

Algorithms
Sorted range algorithms

`binary_search()`	Return whether a collection contains an element
`includes()`	Return whether a collection is a subset of another one
`lower_bound()` `upper_bound()`	Find the first element greater than or equal (strictly greater than) a given value
`equal_range()`	Return the range of elements equal to a given value
`merge()`	Merge two collections
`set_union()` `set_intersection()`	Copy set union (intersection) of two collections into a third one
`set_difference()`	Copy all elements of a collection that are not part of another one into a third collection
`set_symetric_difference()`	Copy all elements that are part of exactly one collection (out of two) into a third collection
`inplace_merge()`	Merge two consecutive collections

Algorithms
Numeric algorithms

- **The "numeric" operations must be defined on the elements**
  - They can be passed as a parameter to the algorithm
  - Or there are default ones

`accumulate()`	Compute the sum (or any other binary operation) of the elements of a collection
`inner_product()`	Scalar product and similar things over two collections
`adjacent_difference()`	Yield a collection consisting of the result of a binary operation (default is difference) on consecutive elements
`partial_sum()`	Yield a collection consisting of the result of a binary operation (default is addition) on all predecessors of each element

Character Strings

- **The notion of a "character" varies**
- **STL character strings are template classes**
  - parameterized by the characteristics of characters, their "traits"
    - representation
    - comparison (collating sequence)
    - copy operations
    - input-output
  - `string ≡ basic_string<char>`
    - This corresponds to the character encoding of the current "locale"

Character Strings
Strings as sequences

- **Strings have all properties of sequences**
  - They look like sort of `vector` of `char` with a different representation
  - Example: converting a character string into an array of characters

```
vector<char> vc;
string s = "hello, world";
copy(s.begin(), s.end(), back_inserter(vc));
```

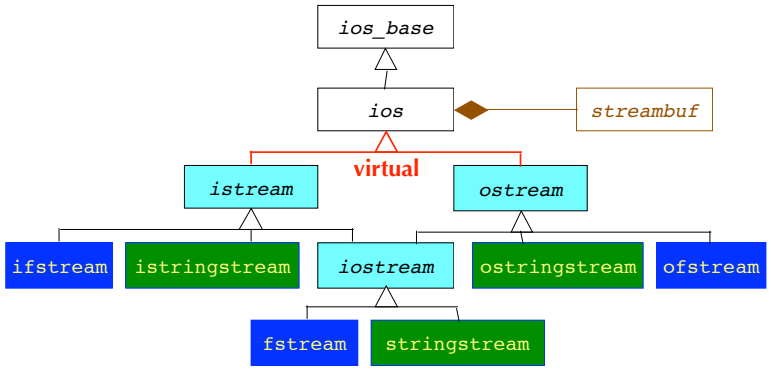
  - Example: removing all blank spaces

```
string s = "some string with spaces";
s.erase(remove(s.begin(), s.end(), ' '), s.end());
```


Input-Output Streams

- **Type driven extensible IO system**
- **Present in C++ nearly from the origin**
- **Improvements in STL**
  - Interface and semantics cleaned up
  - Better extensibility (manipulators)
  - Homogeneity with sequences (iterators)

Input-Output Streams
Classification



Input-Output Streams
Stream iterators

- **Streams, like all containers, can have iterators**

```
vector<string> v;
copy(istream_iterator<string>(cin),
     istream_iterator<string>(),  // EOF
     back_inserter(v));

copy(v.begin(), v.end(),
     ostream_iterator<string>(cout, "---"));
```

  - Note that the default constructor (`istream_iterator()`) corresponds to the end of file

The STL and Exceptions
Exception guarantees (1)

- **STL is not supposed to throw exception for logical problems in containers and algorithms**
  - except for standard exceptions such as `bad_alloc` (impossible to allocate memory)
  - or when explicitly required
    - Only one case: member-function `at()`
  - or user-defined exceptions (in element constructors or operations used by member functions or algorithms)

The STL and Exceptions
Exception guarantees (2)

- **The STL exception guarantees rely on the fact that destructor never throw**
  - either STL own destructors or user destructors
- **The STL offers Basic Guarantee in case of exceptions**
  - No resource leak
  - No violation of container invariants
- **Some containers provide Strong Guarantee (transaction safe operations)**
  - If an operation fails (throws), the container is left in its original state
  - a.k.a. commit-and-rollback guarantee

The STL and Exceptions
Exception guarantees (3)

- **Supplementary guarantees**
  - Node based containers: `list` and associative containers
    - All single element insertions or removals are transaction safe
  - Array based containers: `vector` and `deque`
    - Operations at the end are transaction safe
    - If user copy operations do not throw, all operations are transaction safe
  - `list`
    - All `list` operations are transaction safe...
    - except `remove[_if]`, `sort`, `merge`, and `unique`

The STL and Exceptions
Respect of preconditions (1)

- **Since the STL almost never throws exceptions, all operation calls that do not respect their preconditions result in undefined behaviour**
- **Basic preconditions**
  - Iterators must be valid (next slide)
  - No dereferencing of an iterator that "points to" the end guard
  - Range iterator must be valid
    - Both iterators must point to the same container
    - The second iterator must be reachable from the first
  - If several source ranges are used, they must have at least as many elements as the first one
  - Destination ranges must have enough elements

The STL and Exceptions
Respect of preconditions (2)

- **Iterator validity**
  - An iterator must be initialized before being used
  - As a side-effect of some container operations, iterators can become invalid
  - E.g., for `vector` and `deque`, after insertion and removal, or when reallocation takes place
  - Avoid modifying a container while iterating through it: this does not always crash…

```
list<int> li;  // initially empty
list<int>::iterator it = li.begin();
li.push_back(0);
li.push_back(1);
for ( ; it != li.end(); ++it)
   cout << *it << endl;
// You may be surprised but this loop does nothing!
```


Advanced C++

Annex

Annex
Contents

Annex
Basic C++ concepts and mechanisms

- **Operators `new` and `delete`**
- **Initialization of static data members**
- **A special operator: `operator->`**

C++ Class: Model for Creating instances
Operators `new` and `delete` (1)

- **Dynamically allocated objects is the only kind of objects for which the programmer completely masters lifetime**
- **The programmer has the possibility of redefining or overloading the new and delete operators**
- **Remember that**
  - `new` is just one (out of 4) ways of allocating objects
  - `new` is just a memory allocation operator
    - although, in the case of class instances, the C++ compiler automatically generates a call to the one of the class constructors

	Override	Overload
Global	individual object	individual object
	array	array
Per class	individual object	individual object
	array	array

Redefining `new` and `delete` is tricky, a matter for specialists...

C++ Class: Model for Creating instances
Operators `new` and `delete` (2)

- **Overriding new/delete for individual objects**

```
void *operator new(size_t);      // global operators
void operator delete(void *);
class A {
public:
   // …
   void *operator new(size_t);  // per class, static
   void operator delete(void *);
};
A *pa = new A(...); // use per class
int *pi = new int;  // use global
```

- An idea of generated code (not totally accurate!)

```
// allocation
A *pa = (A *)A::operator new(sizeof(A));
// initialization (call to one constructor)
pa->A::A(...);
```


C++ Class: Model for Creating instances
Operators `new` and `delete` (3)

- **Overriding `new/delete` for arrays**

```
void *operator new(size_t);
int ptab = new string[10]; // use built-in new!!

void *operator new[](size_t);
int ptab = new string[10]; // use new above!!

delete ptab; // use built-in new
void operator delete[](void *);
delete[] ptab; // use delete above
```

C++ Class: Model for Creating instances Operators `new` and `delete` (4)

- **Overloading operator new**

```
void *operator new(size_t, T1, T2, ..., Tn);
A *pa = new (t1, t2, ..., tn) A();
```

- **Operator `new` with pre-placement (predefined)**

```
#include <new>
void *operator new(size_t, void *);

char buff[1000];
A *pa = new (buff) A();
```

  - The object is allocated at the given address (second parameter)
  - The programmer is responsible for allocating storage

C++ Class: Model for Creating instances Operators `new` and `delete` (5)

- **Overloading operator `delete`**
  - For a class only

```
void operator delete(void *, size_t);
```

    - The second parameter is the size to deallocate; it is optional
    - It is automatically filled by the compiler
  - For a class or globally

```
void operator delete(void *, T1, T2, ..., Tn);
```

    - The extra parameters must be identical to the ones of an overloaded `new`
    - This operator is called only from a `new` expression using the overloaded `new`, when the object constructor throws an exception
    - It cannot be called explicitly using a `delete` expression

C++ Class: Model for Creating instances Initialization of static class members (1)

- **Beware: initialization of static data members**

```
// file A.h

class A {
public:
  static string _st;

  static string& fst() {
    static string _fst = "hello";
    return _fst;
  }
};
```

*Simple **declaration**.*
No initialization is possible here

*Local static **definition**.*
Initialization (constructor call) OK

```
// file A.cpp
#include "A.h"

string A:: st("hello");
```

*Static **definition***
Initialization (constructor call) OK

C++ Class: Model for Creating instances Initialization of static class members (2)

- **No guaranteed initialization order across compilation units**

```
// file A.h

class A {
public:
  static string _st;

  static string& fst() {
    static string _fst = "hello";
    return _fst;
  }
};
```

```
// file A.cpp
#include "A.h"

string A::_st = "hello";
```

```
// file B.h

class B {
private:
  static B _bst1;
  static B _bst2;
public:
  B(string);
};
```

```
// file B.cpp
#include "A.h"
#include "B.h"

B B::_bst1(A::_st);     // ??
B B::_bst2(A::fst());   // OK
// ...
```

C++ Class: Abstract Data Type
A special operator: `operator->` (1)

- **Why is `operator->` special?**
  - Overloading an operator means specifying its run time semantics
  - Thus the parameters must be run-time entities
  - But, if we want to keep the natural syntax for `operator->`…

```
A *pa;
pa->member_name(x, y);
bb
```

Not a run time entity!

C++ Class: Abstract Data Type
A special operator: `operator->` (2)

- **Prototype of `operator->`**
  - `operator->` must be a unary member of a class

```
class P {
      // ...
      X operator->() const; // one parameter: this
      // ...
};
```

  - it cannot be a free function
  - it has to return a value (here of some type `X`)

C++ Class: Abstract Data Type
A special operator: `operator->` (3)

- **Usage of `operator->`**
  - If one uses operator-> with a regular pointer argument, the usual (builtin) operator-> is called

```
P *pp; pp->m(); // calls C/C++ regular ->
```

    - `m` must be the name of a member of `P`
  - Otherwise, if one uses operator-> with an instance of or a reference to a class, then this class has to define operator-> and the latter will be called:

```
P p;      // instance
P& rp;    // reference
p->m();   // p.operator->()
rp->m();  // rp.operator->()
```

  - In this case, what is `m`? ...

C++ Class: Abstract Data Type
A special operator: `operator->` (4)

- **Evaluation of `operator->` (in the non-builtin case)**
  - Automatic chaining of `operator->`

```
P p; // p is not a regular pointer
p->m();
```

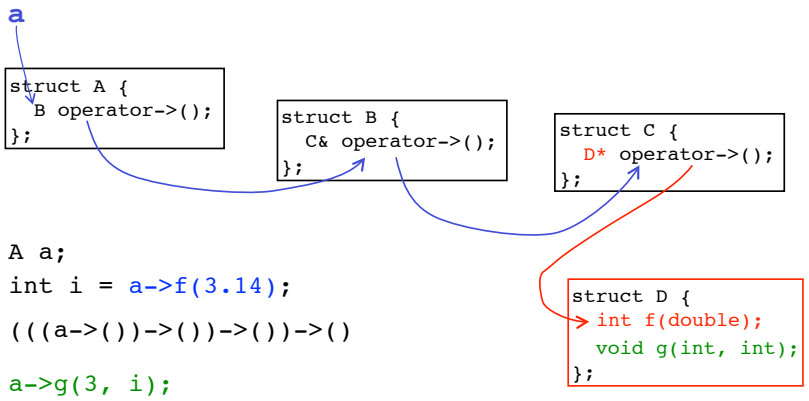
    - evaluate `p.operator->()`, returning an `X` object, say `x`
    - reapply the selection to `x`

```
x->m()
```

    - continue evaluation…

  either `X` is a regular pointer type (`T *`): the evaluation stops selecting `m` in `*x`
  or `X` must be a class defining `operator->` and *the evaluation continue*
  evaluate `x.operator->()`, etc.

C++ Class: Abstract Data Type
A special operator: `operator->` (5)




```
A a;
int i = a->f(3.14);
(((a->())->())->())->()
a->g(3, i);
```


Annex
Advanced C++ Mechanisms

- **Function try-block**
- **Class template and friendship**
- **Template parsing and dependent name look up**
- **Template instantiation and code generation**
- **The power of C++ templates**
- **ISO C++ casts (details)**
- **Miscellanous ISO C++ extensions**
- **Pointers to class members**

Advanced Exceptions
Function try-block (1)

- **Function try block**
  - Particularly useful for constructors (and destructors)

```
B::B(const String& s, int v)
try
  : A1(s), A2(v)
{
  name = s;
  throw Exc_B();
}
catch (Exc_B) {...; cout << s << endl;}
catch (Exc_A2) {...}
```


Advanced Exceptions
Function try-block (2)

- **Function try block (cont. 1)**
  - Beware: special behavior when used for constructors and destructors
  - Base classes and members have been already destroyed when entering the exception handlers (catch clauses)
    - Thus, no reference to base and instance members of the constructed/destructed object is allowed in the handlers
  - The exception caught in the handler is automatically repropagated to the caller
    - Thus, even if the exception has been caught by the constructor/destructor, the object is not considered as properly constructed/destructed

Advanced Exceptions
Function try-block (3)

- **Function try block** (cont.)
  - Extension to regular (and member) functions

```
int f(int x)
try {
  int y;
  // ...
}
catch (E1) { x = 3; return x;} // x visible
catch (E2) { y = 3; return x;} // y not visible
```

  - Here, just a syntactic convenience, no special behavior
  - Note that function parameters (but not function local variables) are visible in the handlers

Complements on Templates
Class template and friendship (1)

- **The friend function does not depend on the template parameters, in any possible way**
  - Friendship with all (visible) instances of the template

```
template <typename T>
class A {
  friend void f();
private:
  int _i;
  // ...
};

void f() {
  A<int> ai;
  A<string> as;
  A<A<double> > aad;
  ai._i = 10;    // OK
  as._i = 10;    // OK
  aad._i = 10;   // OK
}
```


Complements on Templates
Class template and friendship (2)

- **The friend function does depend on (some of) the template parameters**
  - Friendship only with instances of the template with the same effective template arguments

```
template <typename T>
class A {
  friend void f(A<T> a) {
    A<int> ai;
    A<T> at;
    ai._i = 10;   // NO if T
                  // is not int
    a._i = 10;    // OK
    at._i = 10;   // OK
  }
private:
  int _i;
  // ...
};

int main() {
  A<string> as;
  f(as);
}
```


Complements on Templates
Class template and friendship (3)

- **Defining a friend function of a class template**
  - outside the class
    - The function must be declared before the class
    - The friend declaration must specify that a function specialization is intended
    - The friend function may be (often, must be) defined after the class
  - within the class
    - The function is not injected into the global namespace
    - But it will be found by Argument Dependent Lookup (see namespaces, further)

```
template <typename T>
void g(T t);

template <typename T1,
          typename T2>
class A {
  friend void g<T1>(T1);
  friend void h(A a) {
    // ...
  }
};

template <typename T>
void g(T t) {
  // ...
}
```

Complements on Templates
Principle of template parsing

- **Templates are parsed twice**
1. At the template definition
   - Verify basic syntax
   - Look up non-dependent names
2. At instantiation
   - Point of instantiation: where the compiler inserts the substituted template definition
   - Look up dependent names
- *2-phases lookup*

- **Independent name**
  - A name that does not depend on any template parameter
- **Dependent name**
  - A name that explicitly depend on (some) template parameters

```
T::A
A<T>
A<T>::_i
C::m<T>
this
```


Complements on Templates
Dependent name lookup (1)

- **When designating a name inherited from a base class, full qualification must be used**
  - Indeed, in 1st-phase lookup, the compiler cannot bind to the exact base class, because of possible (and yet to come) template specializations of classs A

```
template <typename T>
class A {
protected:
  int _i;
  // ...
};

template <typename T>
class B : public A<T> {
  void f() {
    _i = 12;          // KO
    A<T>::_i = 12;    // OK
    this->_i = 12;    // OK
  }
};
```


Complements on Templates
Dependent name lookup (2)

- **When designating a type nested within a class designated by a dependent name, one must use the `typename` keyword**
  - Indeed, in 1st-phase lookup, the compiler cannot know wether the name effectively designates a type, because of possible (and yet to come) template specializations of classs A

```
template <typename T>
class A {
protected:
  class C {};
  // ...
};

template <typename T>
class B : public A<T> {
  void f() {
    C c; // KO (see before)
    A<T>::C c; // KO
    typename A<T>::C c;// OK
    typename T::iterator it;
};
```


Complements on Templates
Template instantiation code generation (1)

file1.cpp

```
List<int> l1;
List<double> l2;
List<string> l3;
```

file2.cpp

```
List<int> l1;
List<string> l2;
...
```

file3.cpp

```
List<int> l1;
List<double> l2;
...
```

- **How to avoid code duplication?**
  - Each file is compiled separately (compilation unit)
  - Into which object file (`.o`) is the compiler going to generate the template instance binary code?
- **Most systems use *Extended Library Format* (ELF) binaries**
  - The code of the a member-function of a given instance (e.g. List<int>) is generated in each.o file that uses it
  - This code is marked as weak
  - The link editor factorize common weak code

Complements on Templates
Template instantiation code generation (2)

- **Binary code generation: fundamental remark**
  - The automatic template instantiation generates code only for functions and member-functions that are effectively used (instantiated) in the program
- **This is a general rule in C++: *a function/member-function (template or not) that is not effectively used does not need to be defined***
  - There should not be any link edition error (undefined symbol) in this case
  - This does not apply to (non pure) virtual functions that must always be defined
    - it is impossible to know, at compile time, whether they are effectively used

Complements on Templates
Template instantiation code generation (3)

- **Sometimes, it may be desirable to force instantiating a function template or a whole class template**
  - For instance when creating a library containing specific template instances
- **Explicit template instantiation**

```
template class List<int>;
    // generate the whole implementation
template void List<double>::append(const int&);
template int Min<int>(int, int);
     // generate only the mentionned functions
```

  - The binary code is generated in the object file (.o) of the current compilation unit

Complements on Templates
The power of C++ templates

- **TURING completeness of C++ templates**
  - Possibility to compute any Computable Function at compile-time
  - Example: compute the nth term of FIBONNACCI sequence

```
template <int N> struct Fibo {
      static const long long value =
                Fibo<N-1>::value + Fibo<N-2>::value;
}; // Generic form

template <> struct Fibo<0> {
      static const long long value = 1;
  }; // Specialization to stop recursion

template <> struct Fibo<1> {
      static const long long value = 1;
  }; // Specialization to stop recursion

int main() { // we still have to run to print out result!
      cout << Fibo<40>::value << endl;
}
```

- **⇒ template (meta-)programming**

Turing-completeness of C++ Templates (3)

- **Note that the computational complexity of `Fibo<N>` is linear**

Fibo(40)	Compilation	Run
Iterative	0.24	0.00
Recursive	0.24	2.34
Template	0.22	0.00
Intel Xeon at 2.33 GHz (times in seconds)		

ISO C++ Casts
Operator `static_cast`

```
T1 t1;
T2 t2 = static_cast<T2>(t1);
```

- Respect `const` and access control
- If the cast is possible with only static information, it returns the correct value,
  - even in case of regular (non virtual) multiple inheritance
  - this cast must not be used in case of virtual multiple inheritance for downward casting
- This cast is absolutely safe and the result is always correct if the cast forces an **implicit conversion**

ISO C++ Casts
Operator `reinterpret_cast` (1)

```
T1 t1;
T2 t2 = reinterpret_cast<T2>(t1);
```

- `T1` and `T2` may have any pointer type or, more generally, any integral type
- This cast is intrinsically **unsafe**
  - it does not respect access control
  - however, it respects `const`
  - it does not adjust the returned value
- The cast does not change the value, but "reinterpret" it in a new type
- The only sensible use is as a conversion relay
  (convert into some type then, later, back to the original type)

ISO C++ Casts
Operator `reinterpret_cast` (2)

```
// Transform an integer into a pointer
unsigned int addr = 0xffbc4;
int *pi = reinterpret_cast<int *>(addr);

// Transform the type of a pointer
char *buff = new char[BIGSIZE];
int *pi1 = reinterpret_cast<int *>(buff);
int *pi2 =
   reinterpret_cast<int *>(buff + sizeof(int));
```


ISO C++ Casts
Operator `const_cast` (1)

```
void f(const char *s, volatile int *pv) {
  char *pc = const_cast<char *>(s);
  int *pi = const_cast<int *>(pv);
}
```

- This cast removes constness and/or volatility
- The target type must be the original type but without `const` or `volatile`
  - Remember that the reverse conversion is an implicit one, possible at compile-time, hence safe (it is a `static_cast`)
  - If the object is actually constant or actually volatile, the result is undefined!
- This cast should be avoided

ISO C++ Casts
Operator `const_cast` (2)

- **How to avoid `const_cast`?**
- **Mutable members**
  - Non constant members even in constant objects
  - Similar to `const` and `volatile` (but only for class members)

```
class A {
  mutable int refcnt;
  int attr;
  void f() const {
    refcnt++; // modification of refcnt
    attr = 0; // forbidden
  }
  // ...
};
```


Miscellaneous ISO C++ extensions

- **New keywords, digraphs and trigraphs**
- **Really miscellaneous extensions**
- **Explicit constructors**
- **Initialization of `const static` integers**
- **Covariant return**
- **Pointer to class members**

Miscellaneous ISO C++ extensions
Keywords

and	continue	if	register	typedef
and_eq	default	inline	reinterpret_cast	typeid
asm	delete	int	return	typename
auto	do	long	short	union
bitand	double	mutable	signed	unsigned
bitor	dynamic_cast	namespace	sizeof	using
bool	else	new	static	virtual
break	enum	not	static_cast	void
case	explicit	not_eq	struct	volatile
catch	extern	operator	switch	wchar_t
char	false	or	template	while
class	float	or_eq	this	xor
compl	for	private	throw	xor_eq
const	friend	protected	true	
const_cast	goto	public	try	

`struct` already in Ansi C | **`bitand`** specific to C++, strange? | **`class`** specific to C+

Miscellaneous ISO C++ extensions
Digraphs and trigraphs

- **Beware of the following character combinations**
  - Trigraphs are replaced before any source processing
  - A digraph is equivalent to some character, but its spelling is preserved (in particular within character strings)

Trigraphs (Ansi C compatibility)				**Digraphs** (C++ specific)	
Trigraph	Replacement for	Trigraph	Replacement for	Digraph	Equivalent to
??=	#	??!	|	<%	{
??/	\	??<	{	%>	}
??'	^	??>	}	<:	[
??(	[	??-	~	:>	]
??)	]			%:	#

Miscellaneous ISO C++ extensions
Really miscellaneous

- **Support for European and extended characters**
- **If the loop index is declared in the for clause, its scope is limited to the loop body (including the for clause)**

```
for (int i = 0; i < N; i++)
 if (t[i] == x) break;
t[i]++;  // NO: i is unknown here
```

- **In the same spirit, variables may be defined in conditions**

```
if (int c = ...) {/* c is known here */}
while (int c = ...) {/* c is known here */}
```


Miscellaneous ISO C++ extensions
Preventing constructor implicit conversion

- **Explicit constructors (explicit)**

```
class A {
public:
 explicit A(int);
 // ...
};
a = 5;   // implicit user conversion A(5)
forbidden
A a = 3; // NO
A a(3);  // OK
A a = A(3); // a complicated form for the previous
a = A(5);// OK
```


Miscellaneous ISO C++ extensions
Initialization of const static integer members

- **Direct initialization of const static integer members**

```
class Table {
  const static int N = 10;// OK
  int t[N];  // OK
};
```

  - These are the only members that can be initialized "on the fly"
  - Note that they must be at the same time `const`, `static`, and of some integral type
    - `char`, `short`, `long`, `long long` and their unsigned variations, `enum`
    - but not pointers

Miscellaneous ISO C++ extensions
Covariant return

- **When overriding a member-function in a derived class, it is possible to restrict its return type (covariance)**

```
class T {
  // ...
};

class U : public T {
  // ...
};
```

```
class A {
  T f(int, double);
};

class B : public A {
  U f(int, double);
};
```

Pointers to Class Members
Pointer to member-functions (1)

- **Define a function taking a member-function as parameter**

```
class Window {
  int width, height;
  int cursor_x, cursor_y;
public:
  void up();
  void down();
  void right();
  void left();
  void repeat( ??? , int n);
};
```


Pointers to Class Members
Pointer to member-functions (2)

- **What is the type of up, down... ?**
  - Certainly not void (*)() because class membership must be taken into account
  - Pointer to a member-function of class Window with no parameter nor result:

```
void (Window::*)()
```


Pointers to Class Members
Pointer to member-functions (3)

- **Using a pointer to member through a pointer**

```
void Window::repeat(void (Window::*op)(), int n) {
 for (int i = 0; i < n; ++i)
    (this->*op)();
}
```

- **Using a pointer to member through an instance/reference**

```
void afunction(void (WIndow::*op)(), Window& w) {
 (w.*op)();
}
```

- **Typedef'ing a pointer to member**

```
typedef void (Window::*Movements)();
Movements tab[] = {&Window::up,
```


Pointers to Class Members
Pointer to data members

```
class A {
  int x;
  int y;
  // ...
};

int A::*pmi = &A::x;
A a, *pa = &a;
a.*pmi = 12;      // a.x = 12
pmi = &A::y;
pa->*pmi = 27: // pa->y = 17
```

Pointers to Class Members
Standard conversion of pointers to members

- **A pointer to a base class member can be implicity converted into a pointer to a derived class member**

```
class A {
  int x;
  // ...
};
class B : public A { ... };

int A::*pma = &A::X;
B b;
b.*pma = 12;   // OK: b.x ≡ b.A::x = 12
```

- Note that this conversion goes into the direction opposite to the other inheritance conversions

Pointers to Class Members
Pointers to members properties

- **Pointers to members are sort of *portable offsets***
  - They take care of pointer/offset adjustment in case of multiple (virtual) inheritance
- **Pointers to member-functions always honor virtuality (dynamic binding)**

Annex
The Standard Template Library (STL)

- **Common and specific members of STL containers**
- **Boost and TR1 `bind()` adaptor and `function<...>`**
- **Header files (C++03 library)**

Container Members
Common Members (1)

Member types	
Cont`::value_type`	type of element
Cont`::iterator`	≈ `value_type*`
Cont`::const_iterator`	≈ `const value_type*`
Cont`::reverse_iterator`	≈ `value_type*`
Cont`::const_reverse_iterator`	≈ `const value_type*`
Cont`::reference`	≈ `value_type&`
Cont`::const_reference`	≈ `const value_type&`
Cont`::allocator_type`	type of memory allocator

Container Members
Common Members (2)

Contruction and destruction	
`Cont c;`	Create an empty container
`Cont c1(c2);`	Create a container from another container with the same value type
`Cont c(beg,end);`	Create a container by copying the values in `[beg, end[`
`c.~Cont()`	Destructor
Size operations	
`c.size()`	Number of elements in the container
`c.empty()`	True if the container contains no elements
`c.max_size()`	Maximum possible number of elements

Container Members
Common Members (3)

Comparisons	
`c1 == c2`	Are the two containers equal (all their elements are pairwise equal)
`c1 != c2`	Equivalent to !(c1 == c2)
`c1 < c2` (and `<=`, `>`, `>=`)	Lexicographic comparison of containers (the value types must be identical and it must have `operator<()`)
Assignments	
`c1 = c2`	Copy assignment
`c1.swap(c2)`	Swap internal data of c1 and c2; the two containers must have the same type
`swap(c1,c2)`	Equivalent to c1.swap(c2)

Container Members
Common Members (4)

Iterator interface	
`c.begin()`	"Point" to first element
`c.end()`	"Point" to just after last element
`c.rbegin()`	"Point" to first element in reverse order
`c.rend()`	"Point" to just after last element in reverse order
Insertion and removal	
`c.insert(pos,e)`	Insert element of value `e`. The return value and the interpretation of pos depend on the container type
`c.erase(beg,end)`	Remove all elements in the range `[beg, end[`
`c.clear()`	Remove all elements (revert to empty container)

Container Members
Specific members: `vector`, `deque`, `list`

Construction		v	dq	l
`Cont` `c(n);`	Create a container with `n` elements initialized to the zero of their type	✓	✓	✓
`Cont` `c(n,e);`	Create a container with `n` elements initialized to value `e`	✓	✓	✓
`c.capacity()`	Total number of elements possible without reallocation	✓	✓	✗
`c.reserve(n)`	Reserve room for at least `n` elements	✓	✓	✗
Set values		**v**	**dq**	**l**
`c.assign(n)`	Assign the first `n` elements to the zero of their type	✓	✓	✓
`c.assign(n,e)`	Assign the first `n` elements to value `e`	✓	✓	✓

Container Members
Specific members: `vector`, `deque`, `list`

Access to elements		v	dq	l
`c.at(i)`	Return a reference to element at index `i`. If it does not exist, throw `out_of_range`	✓	✓	✗
`c[i]`	Return a reference to element at index `i`. No check	✓	✓	✗
`c.front()`	Return a reference to first element. No check	✓	✓	✓
`c.back()`	Return a reference to last element. No check	✓	✓	✓
Resize		**v**	**dq**	**l**
`c.resize(n)`	Change size to `n`; if size grows new elements are initialized to the zero of their type	✓	✓	✓
`c.resize(n,e)`	Change size to `n`; new elements initialized to `e`	✓	✓	✓

Container Members
Specific members: `vector`, `deque`, `list`

Insertion and removal		v	dq	l
`c.insert(pos,n,e)`	Insert `n` times value `e` at position given by iterator `pos` (that is before)	✓	✓	✗
`c.insert(pos,beg,end)`	Insert elements in range `[beg,end[` at position `pos`	✓	✓	✗
`c.push_back(e)`	Append element `e`	✓	✓	✓
`c.pop_back()`	Remove last element. Return nothing. No check	✓	✓	✓
`c.push_front(e)`	Prepend element `e`	✗	✓	✓
`c.pop_front(e)`	Remove last element. Return nothing. No check	✗	✓	✓
`c.erase(pos)`	Remove element at position `pos`. Return an iterator to the successor	✓	✓	✓

Container Members
Specific members: `list`

Splice operations	
`c.unique()`	Remove duplicate consecutive values
`c.unique(op)`	Remove duplicate consecutive elements for which `op()` is true
`c1.splice(pos,c2)`	Move element at `pos` in `c2` in front of element at `pos` in `c1`; `c1` and `c2` may be the same list
`c1.splice (pos,c2,beg,end)`	Move elements in the range `[beg,end[` in `c2` in front of element at pos in `c1`; `c1` and `c2` may be the same list
`c.sort()`	Sort elements with `operator<()`
`c.sort(op)`	Sort elements with `op()`
`c1.merge(c2)`	Merge elements of `c2` into `c1`; `c1` and `c2` must be sorted
`c1.merge(c2,op)`	Same as above with `op()` instead of `operator<()`
`c1.reverse()`	Reverse list `c1`

Container Members
Specific members: `stack` and `queues`

Insertion and removal		s	q	pq
`c.push(e)`	Add an element with value `e`	✓	✓	✓
`c.pop()`	Remove an element	✓	✓	✓
`c.top()`	Return a reference to first element. No check	✓	✗	✓
`c.front()`	Return a reference to first element. No check	✗	✓	✗
`c.back()`	Return a reference to last element. No check	✗	✓	✗

- **Internal sequence container is a `deque` by default for `stack` and `queue`, a `vector` for `priority_queue`**
- **For priority queues, sorting is in decreasing order w.r.t `operator<()`**
  - This can be changed with a template parameter

Container Members
Specific members: associative containers (1)

Construction	
`Cont c(op);`	Create an empty container with `op()` as sorting criterion
`Cont c(beg,end,op);`	Create a container by copying elements in range `[beg,end[` with `op()` as sorting criterion
Accessors	
`c.count(k)`	Return number of occurrences of `k`
`c.find(k)`	Return an iterator on cell with `k`
`c.lower_bound(k)`	Return an iterator to the first possible location for value `k`
`c.upper_bound(k)`	Return an iterator to the last possible location for value `k`
`c.equal_range(k)`	Return a pair of iterators to the possible range for value `k`

Containers and Iterators
Specific members: associative containers (2)

Insertion and removal	
`c.insert(k)`	Insert element `k`, return iterator to position for `multiset` and `multimap` and a `pair<iterator, bool>` for `set` and `map`
`c.insert(beg,end)`	Insert all elements in range `[beg,end[`. Return nothing
`c.erase(k)`	Erase all elements with value `k`
`c.erase(pos)`	Erase the element at iterator position `pos`. Return nothing

- **Note that `k` is a constant for all associative containers**
  - You cannot change the value of the key, since it is used to sort the elements

High Order Programming
Boost and TR1 `function` (1)

- **Problem with `bind`**
  - Creating and storing a function object to use it later?

```
void foo(int a, int b) { . . . }
??? fobj = bind(foo, _1, 3); // type unknown
fobj(5); // ???
```

    - The type is determined with respect to the *context of call*
- **Use Boost or TR1 `function`**

```
function<void(int)> fobj = bind(foo, _1, 3);
fobj(5);
```

  - A powerful replacement for *pointer to functions*...

High Order Programming
Boost and TR1 `function` (2)

- **Construction of function objects using function**

```
function<int(int, int)> f;
```

  - The default constructor yields an empty function object

```
if (f.empty()) . . . // true
if (not f) . . .     // true
f(3, 3);             // throw bad_function_call
```

  - The function object can be initialized with
    - a regular function or a function object of the same signature
    - a reference wrapper to a function object of the same signature
    - a member function the first parameter of which must be the class of this member
  - The function objects created with function are copiable
  - Up to 10 parameters (library configuration)

High Order Programming
Boost and TR1 `function` (3)

- **Examples of construction of function objects using `function`**

```
int myplus(int, int);
function<int(int, int)> f1 = &myplus;
function<int(int, int)> f2 = plus<int>();
int a = f1(3, 4);
int b = f2(a, 5);
f1 = f2;

struct Figure {
  double rotate(double);
  // . . .
};
function<double(Figure, double)> f3 = &Figure::rotate;
function<double(Figure*, double)> f4 = &Figure::rotate;
double x = f3(Figure(), 3.5);
x = f4(new Figure(), 3.5);
```


High Order Programming
Boost and TR1 `function` (4)

- **Function objects with state**
  - As usual, function objects may have internal attributes
  - Thus they accumulate information each time they are called
- **However always remember that function objects are copied by value**

```
struct Accumulate {
  int _total;
  Accumulate() : _total(0) {}
  int operator()(int a) {
    _total += a;
    return _total;
  }
};

Accumulate acc;
function<int(int)> facc1 = acc;
function<int(int)> facc2 = acc;
cout << facc1(10) << ' ' << facc2(10) << endl; // 10 10
cout << acc._total << endl; // 0
```


High Order Programming
Boost and TR1 `function` (4bis)

- **Function objects with state**
  - As usual, function object may have internal attributes
  - Thus they accumulate information each time they are called
- **However always remember that function objects are copied by value**

```
struct Accumulate {
  int _total;
  Accumulate() : _total(0) {}
  int operator()(int a) {
    _total += a;
    return _total;
  }
};

Accumulate acc;
function<int(int)> facc1 = ref(acc); // form a reference
function<int(int)> facc2 = ref(acc); // form a reference
cout << facc1(10) << ' ' << facc2(10) << endl; // 10 20
cout << acc._total << endl; // 20
```


High Order Programming
Boost and TR1 `function` (5)

- **`function` and `bind`**
  - These two functions are compatible
    - One can create a `function<...>` as the result of `bind`

```
function<int(int)> f1 = bind(plus<int>(), _1, 3);
```

    - One can also bind the arguments of a `function<...>`

```
function<int()> f2 = bind(f1, 5);
```

High Order Programming
Boost and TR1 `function` (6)

```
struct Window {
  int _x, _y;
  Window() : _x(0), _y(0) {}
  void up();
  void down();
  void left();
  void right();
};

int main()
{
  typedef function<void()> Command;
  Window w;
  map<string, Command> cmds = {    // C++11 initialization syntax
    {"up",    bind(&Window::up, ref(w))},
    {"down",  bind(&Window::down, ref(w))},
    {"left",  bind(&Window::left, ref(w))},
    {"right", bind(&Window::right, ref(w))},
  };

  string cmd;
  while (cin >> cmd)
    cmds.at(cmd)();     // throw exception 'out_of_range' if cmd invalid
}
```


High Order Programming
Boost and TR1 `function` (7)

- **Advantages of `function`**
  - Replacement of pointers to functions
  - Compatible with (nearly) all ways of representing something like a function in C++
  - Compatible with STL 2003 for 1 and 2 argument functions
  - Function objects may have state
- **Drawbacks**
  - Be careful with copying function objects
  - Cost: 4 times bigger than a regular pointer to function
  - Does not solve the overloading problem

```
int g(int i);
double g(double x);
function<int(int)> fg = &g; // does not compile
function<int(int)> fg = static_cast<int(*)(int)>(&g);
```


High Order Programming
Boost and TR1 `mem_fn` adaptor

- **`mem_fn`, a replacement for `mem_fun` and `mem_fun_ref`**
  - Works for **member fonctions**
  - Number of parameters up to 9 (plus `this`)
  - No more distinction between collection of pointers or of objects
  - Compatible with STL 2003 function objects and functionals
    - The function object created by `mem_fn` derives from `unary_function` or `binary_function` (when the number of parameters is adequate)
  - Compatible with `bind` (although not often used)
    - since `bind` achieves `mem_fn` functionality most of the time

Header files (C++03 library) (1)

- **Containers**

`<vector> <list> <deque>`
`<queue> <stack> <map>`
`<set> <bitset>`

- **General utilities**

`<utility> <functional>`
`<memory> <ctime>`

- **Iterators**

`<iterator>`

- **Algorithms**

`<algorithm> <cstdlib>`

- **Diagnostics**

`<stdexcept> <cassert>`
`<cerrno>`

- **Strings**

`<string> <cctype>`
`<cwtype> <cstring>`
`<cwstring> <cstdlib>`

Header files (C++03 library) (2)

Input-output

```
<iosfwd> <iostream>
<ios> <streambuf>
<istream> <ostream>
<iomanip> <sstream>
<cstdlib> <fstream>
<cstdio> <cwchar>
```

Localization

```
<locale> <clocale>
```

Run-time support

```
<limits> <climits>
<cfloat> <new>
<typeinfo> <exception>
<cstddef> <cstdarg>
<csetjmp> <cstdlib>
<ctime> <csignal>
```

Numerics

```
<complex> <valarray>
<numerics> <cmath>
<cstdlib>
```


Some references

- **International Standard for Information Systems — Programming Language C++**
  ISO/IEC 14882:2003 and ISO/IEC 14882:2011
- **The C++ Programming Language** (3rd Special Edition),
  Bjarne STROUSTRUP, Addison Wesley, 2000
- **Programming: Principles and Practice Using C++**
  Bjarne STROUSTRUP, Addison Wesley, 2008
- **The Design and Evolution of C++**
  Bjarne STROUSTRUP, Addison Wesley, 1994
- **Advanced C++ — Programming Styles and Idioms**
  James O. COPLIEN, Addison Wesley, 1992
- **The C++ Standard Template Library**
  Nicolaï M. JOSUTTIS, Addison Wesley, 1999
- **C++ Templates — The Complete Guide**
  David VANDEVOORDE and Nicolai JOSUTTIS, Addison Wesley, 2003
- **Effective C++ CD — 85 Specific Ways to Improve your Program Design**
  Scott MEYERS, Addison Wesley, 2000
- **Effective STL — 50 Specific Ways to Improve your Use of the Standard Template Library**
  Scott MEYERS, Addison Wesley, 2001
- **Modern C++ Design — Generic Programming and Design Patterns Applied**
  Andreï ALEXANDRESCU, 2001